

(12) **United States Patent**
Farrell et al.

(10) **Patent No.:** **US 9,412,195 B2**
(45) **Date of Patent:** **Aug. 9, 2016**

(54) **CONSTANT BUFFER SIZE MULTI-SAMPLED ANTI-ALIASING DEPTH COMPRESSION**

(71) Applicant: **Intel Corporation**, Santa Clara, CA (US)

(72) Inventors: **Robert L. Farrell**, Granite Bay, CA (US); **Prasoonkumar Surti**, Folsom, CA (US); **Vasanth Ranganathan**, El Dorado Hills, CA (US); **Karol A. Szerszen**, Rancho Cordova, CA (US)

(73) Assignee: **Intel Corporation**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **14/492,173**

(22) Filed: **Sep. 22, 2014**

(65) **Prior Publication Data**

US 2016/0086374 A1 Mar. 24, 2016

(51) **Int. Cl.**
G06T 15/40 (2011.01)
G06T 5/00 (2006.01)
G06T 9/00 (2006.01)

(52) **U.S. Cl.**
CPC **G06T 15/405** (2013.01); **G06T 5/002** (2013.01); **G06T 9/00** (2013.01)

(58) **Field of Classification Search**

None

See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

6,226,000 B1 *	5/2001	Richens	G06T 15/04
				345/419
6,246,415 B1 *	6/2001	Grossman	G06T 15/405
				345/421
6,549,203 B2 *	4/2003	Randel	G06T 15/506
				345/426
6,577,307 B1 *	6/2003	Hsiao	G06T 15/503
				345/421

* cited by examiner

Primary Examiner — Said Broome

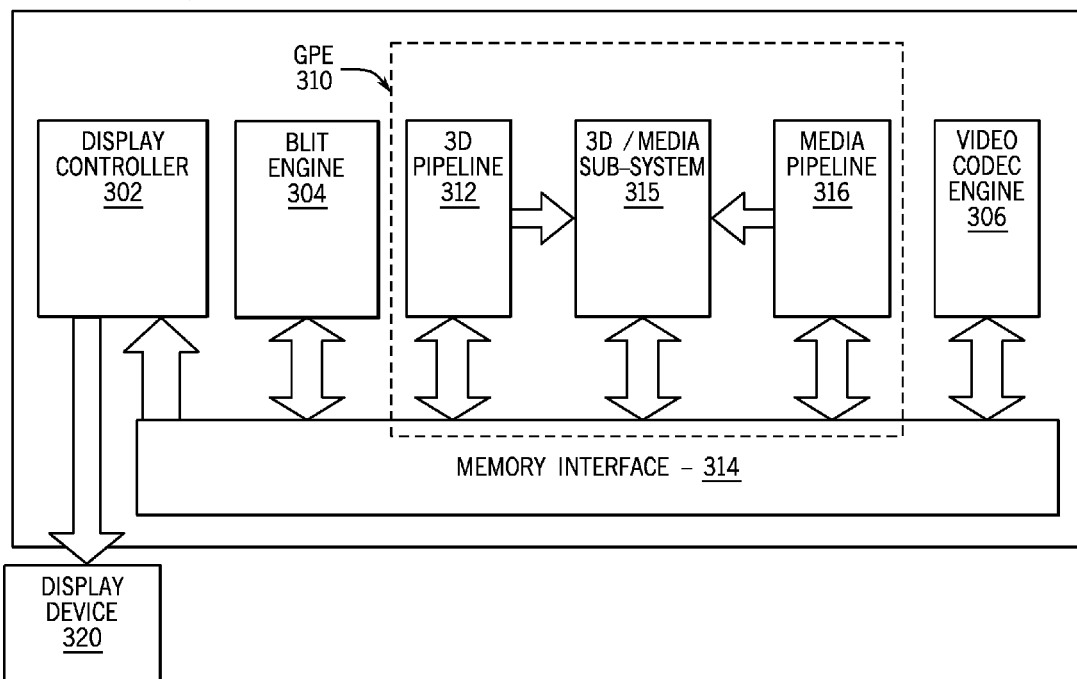
(74) *Attorney, Agent, or Firm* — Trop Pruner & Hu, P.C.

(57) **ABSTRACT**

By packing the depth data in a way that is independent of the number of samples, so that memory bandwidth is the same regardless of the number of samples, higher numbers of samples per pixel may be used without adversely affecting buffer cost. In some embodiments, the number of pixels per clock in a first level depth test may be increased by operating in the pixel domain, whereas previous solutions operated at the sample level.

12 Claims, 15 Drawing Sheets

GRAPHICS PROCESSOR
300



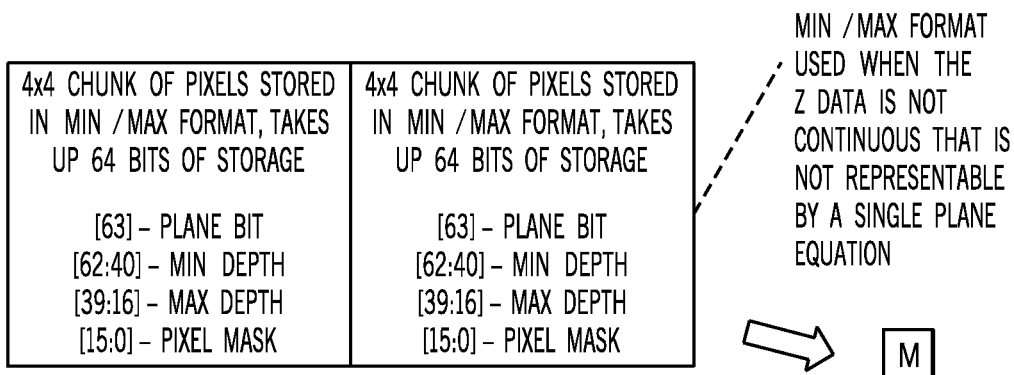


FIG. 1

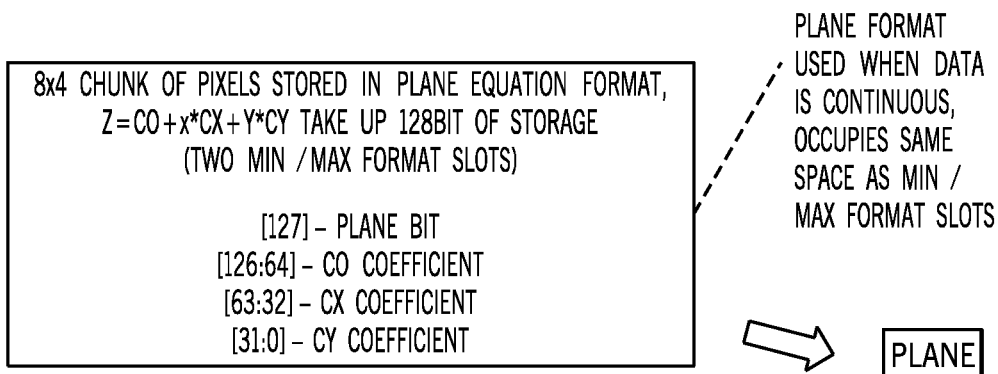


FIG. 2

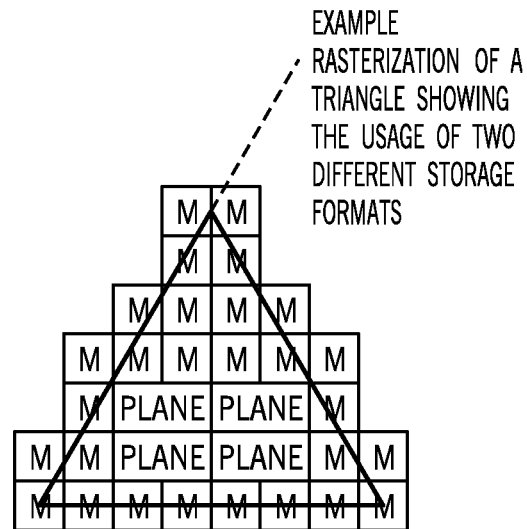


FIG. 3

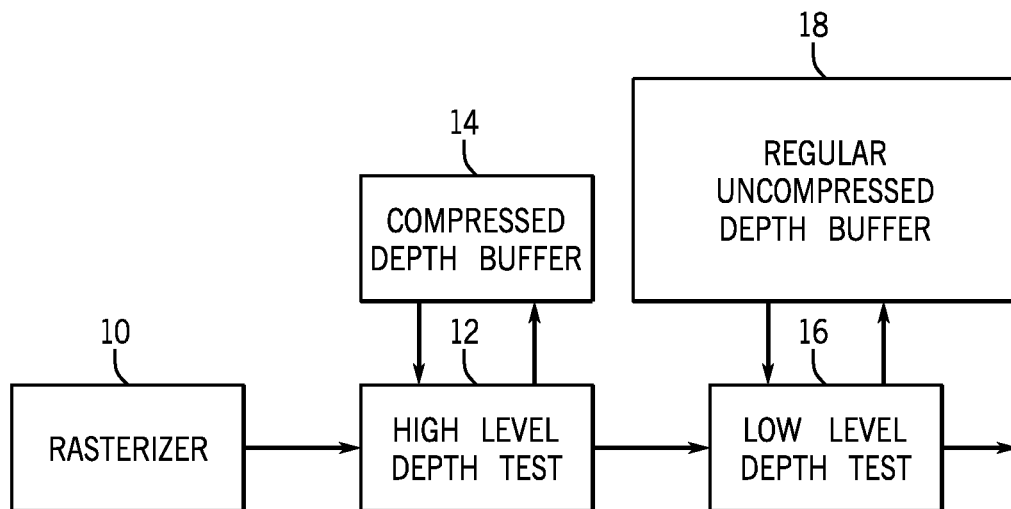


FIG. 4

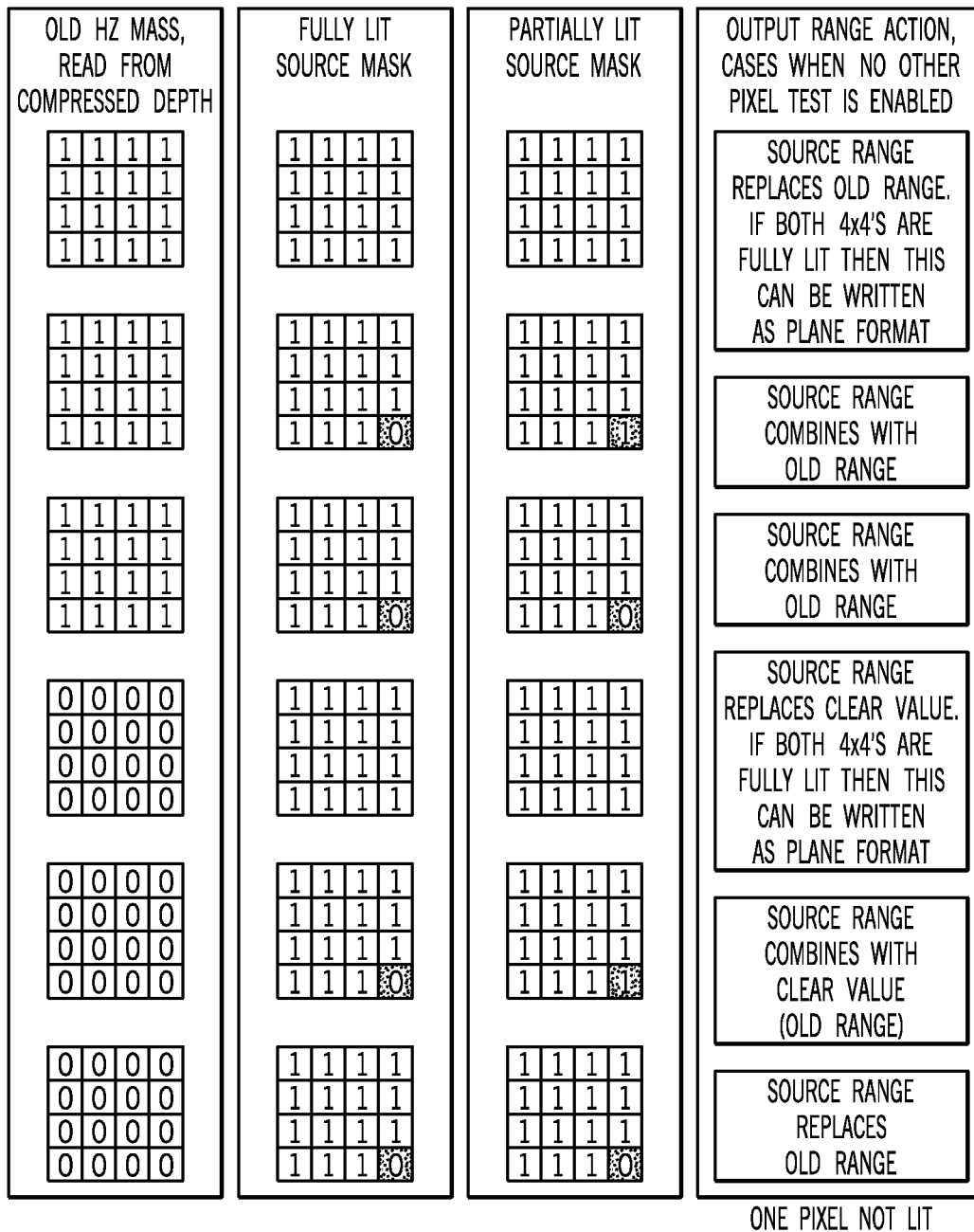


FIG. 5

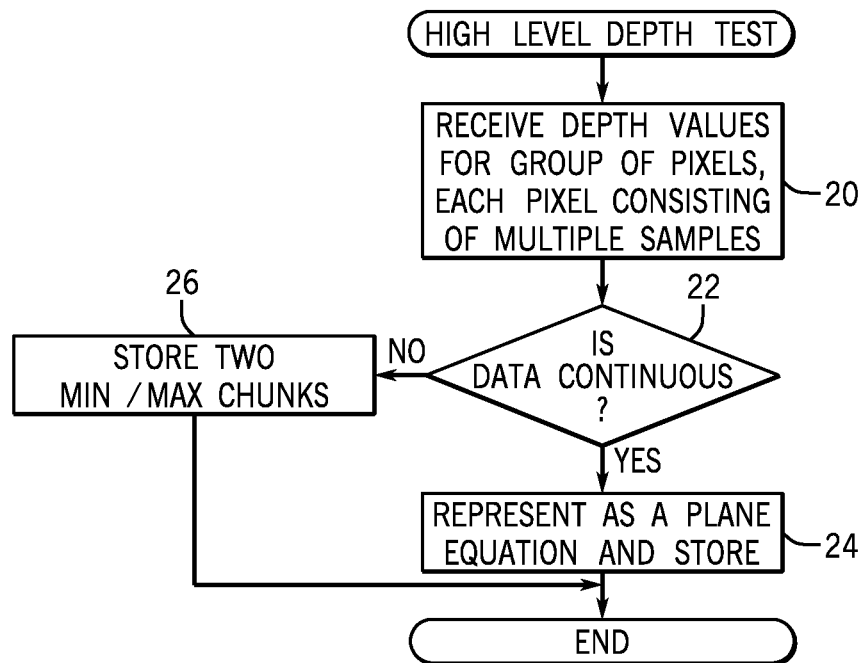
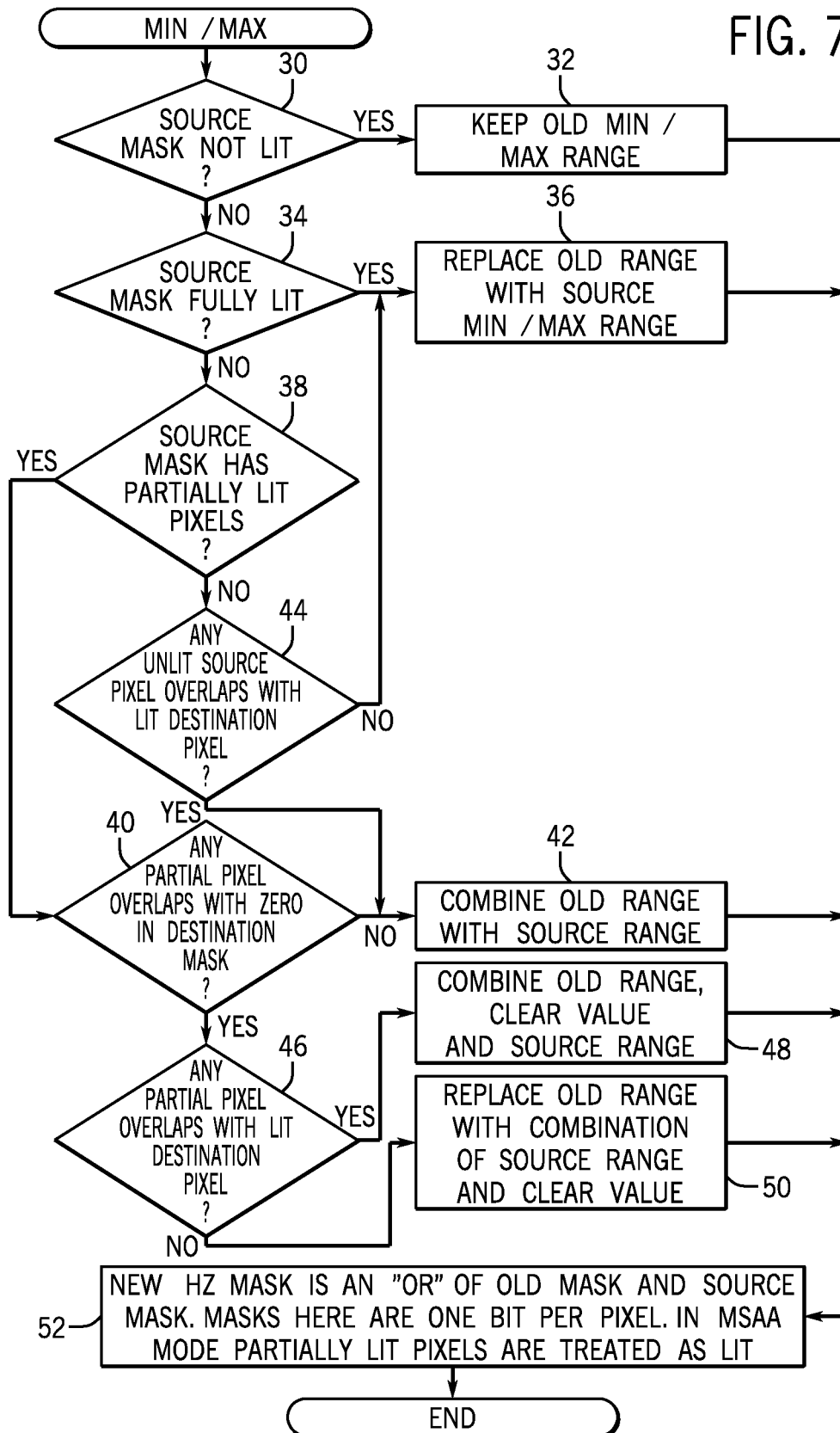


FIG. 6

FIG. 7



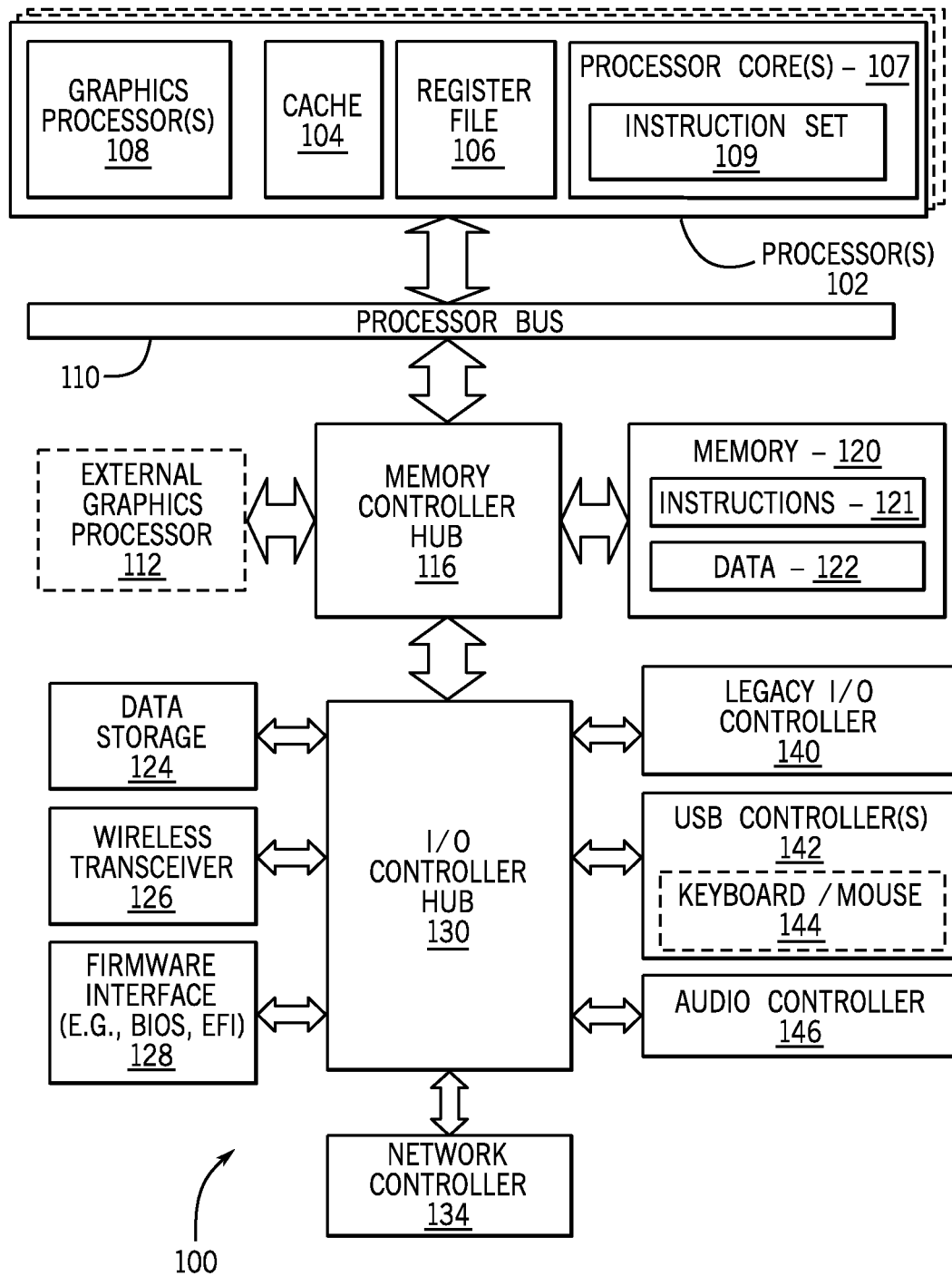


FIG. 8

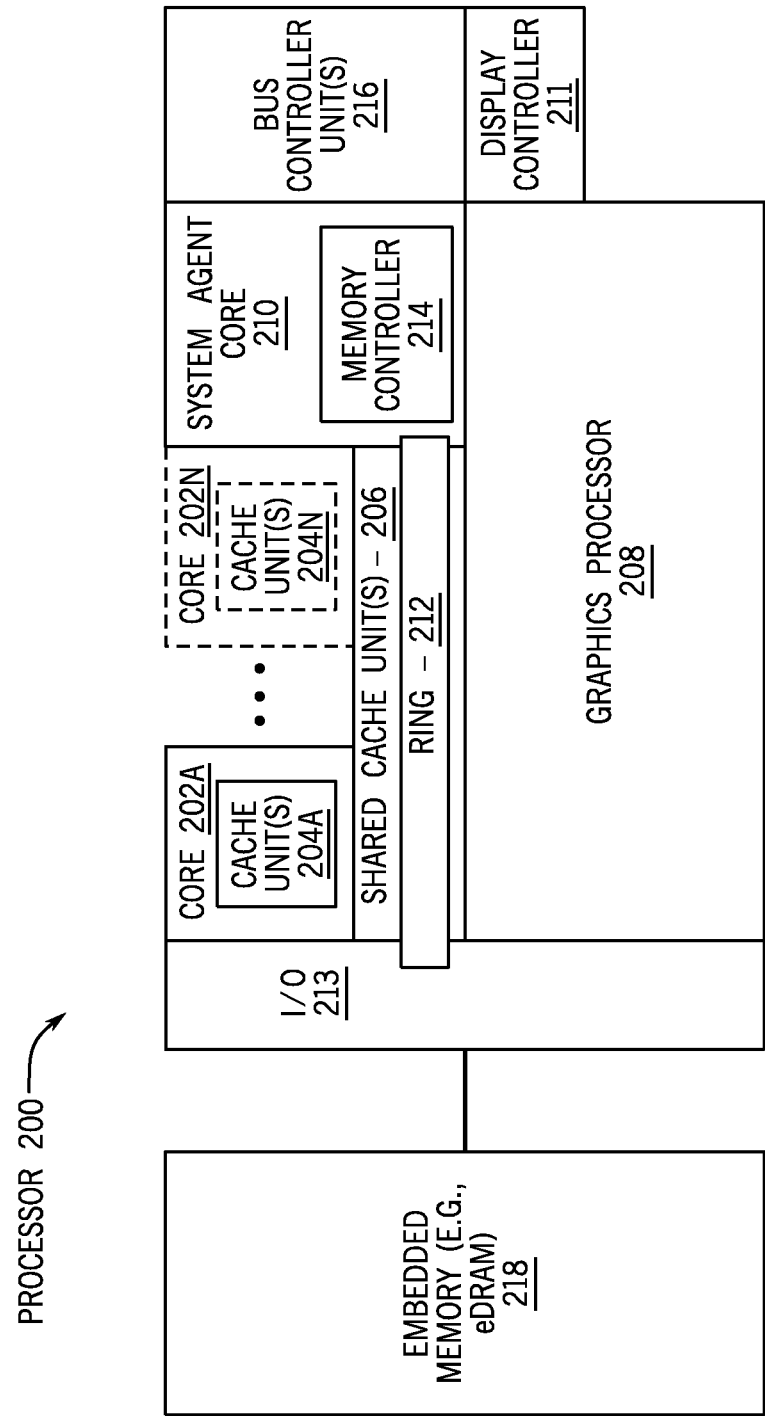


FIG. 9

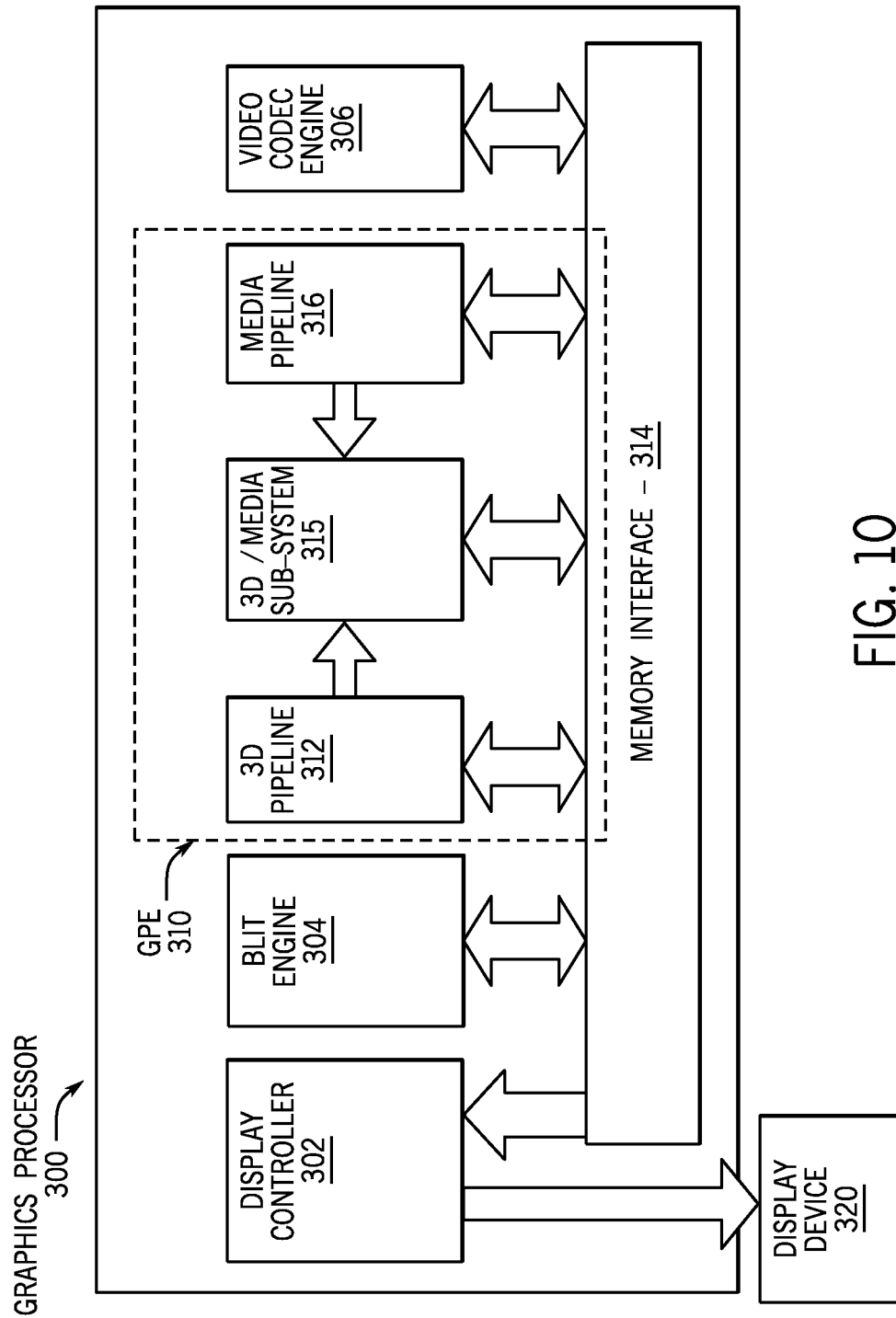


FIG. 10

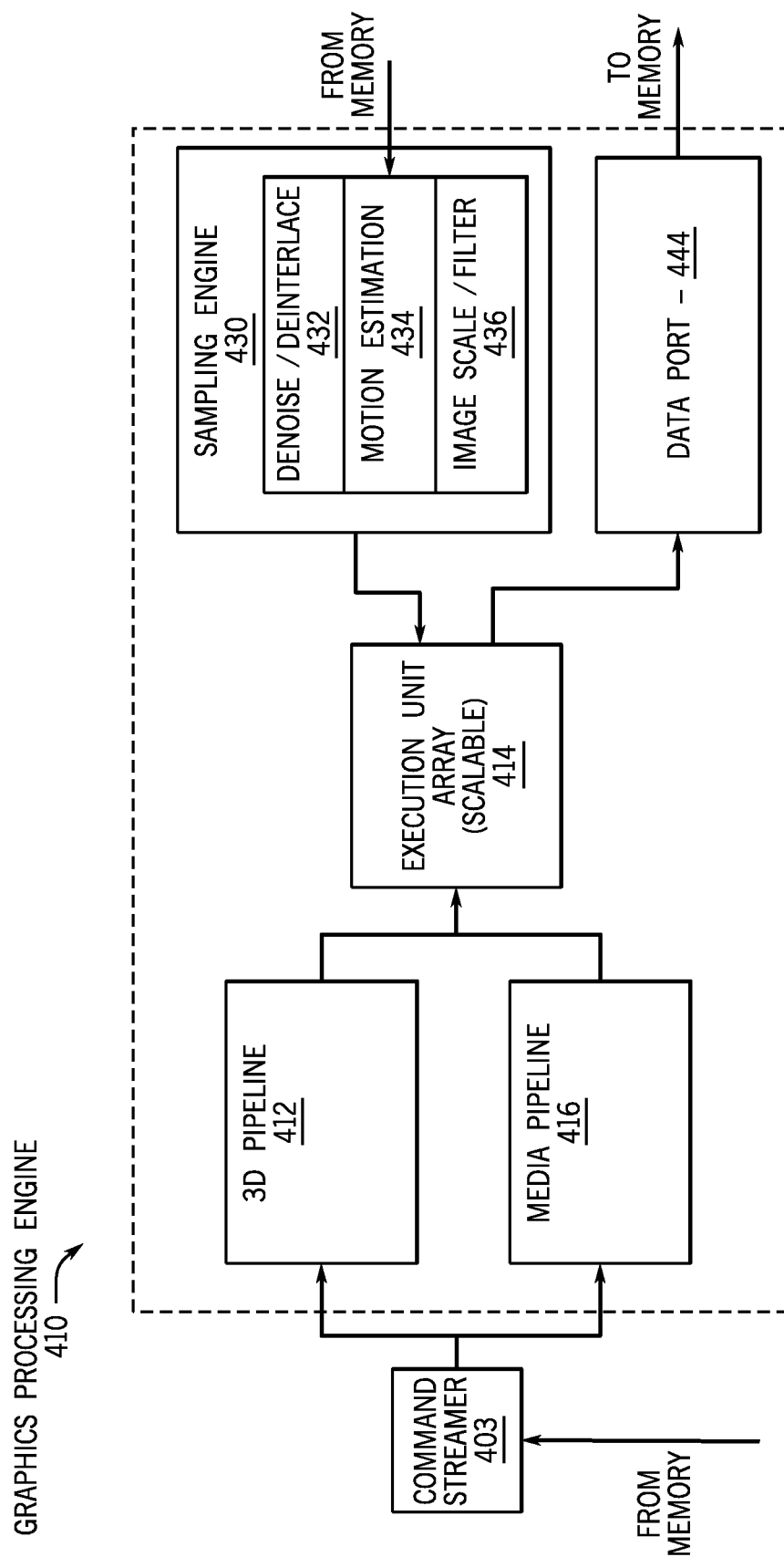


FIG. 11

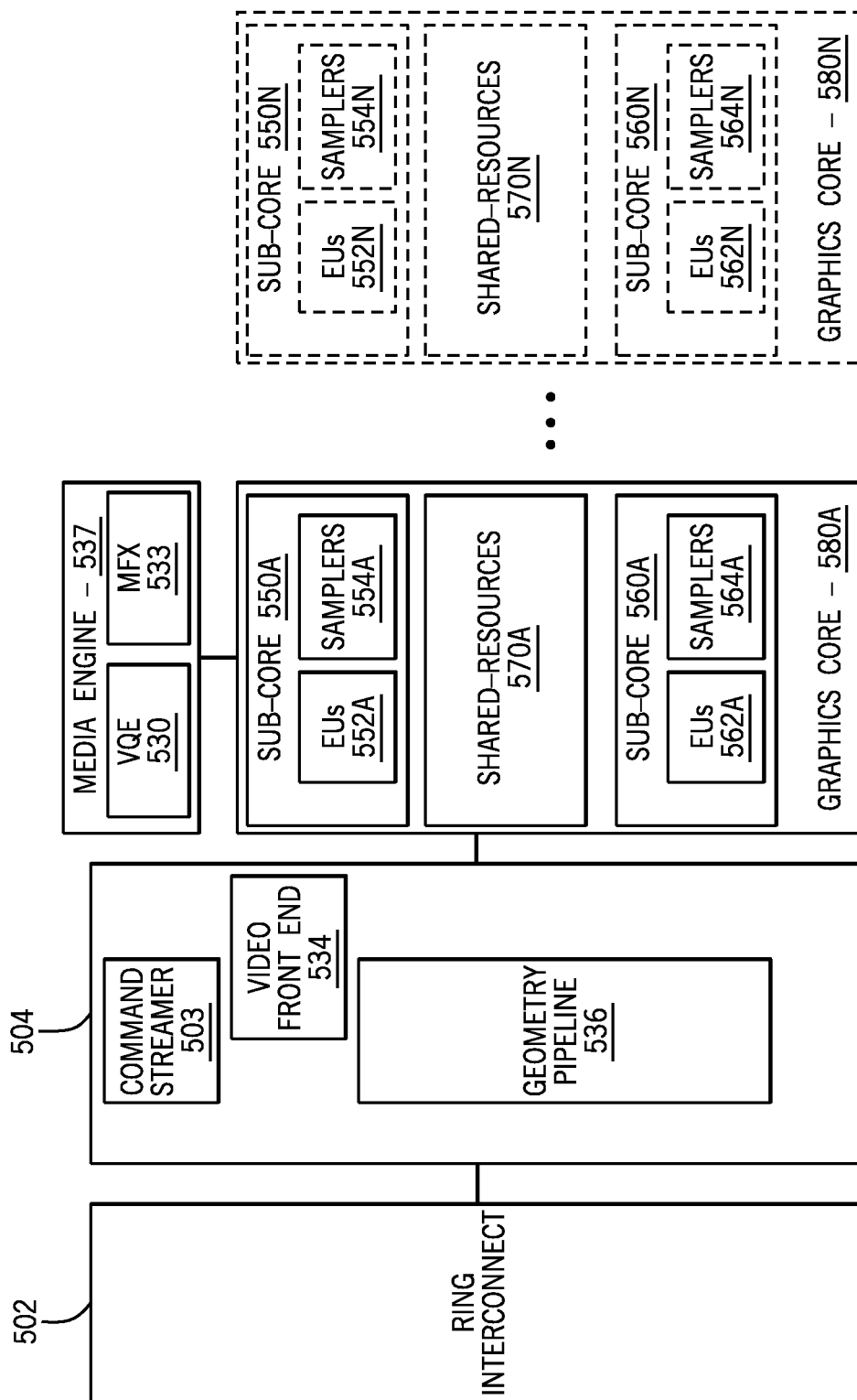


FIG. 12

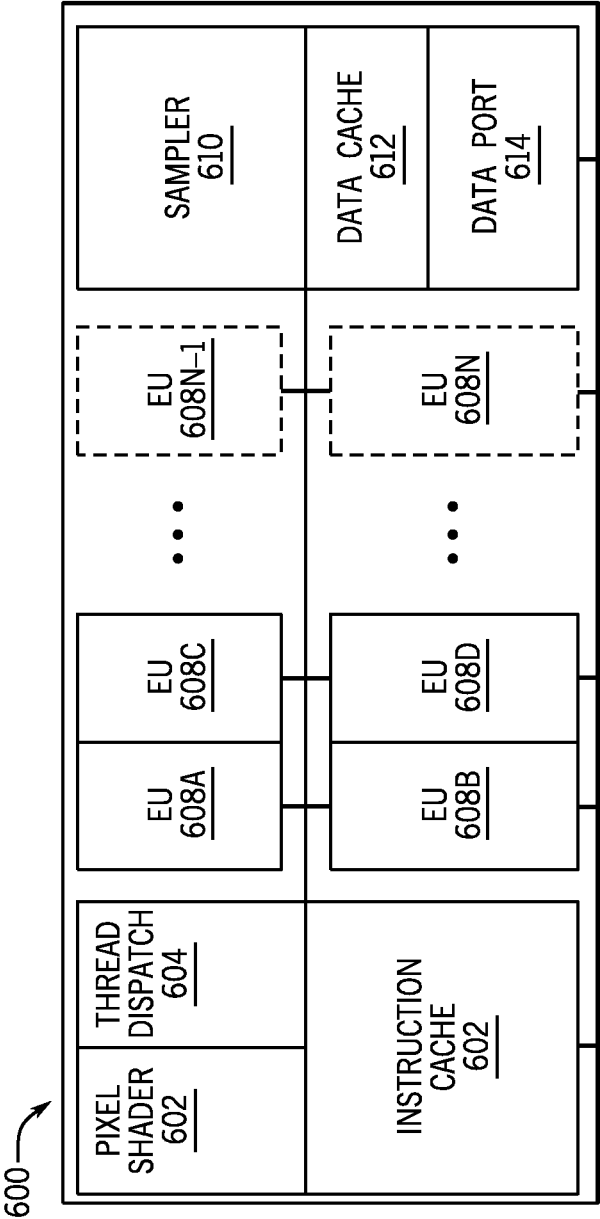
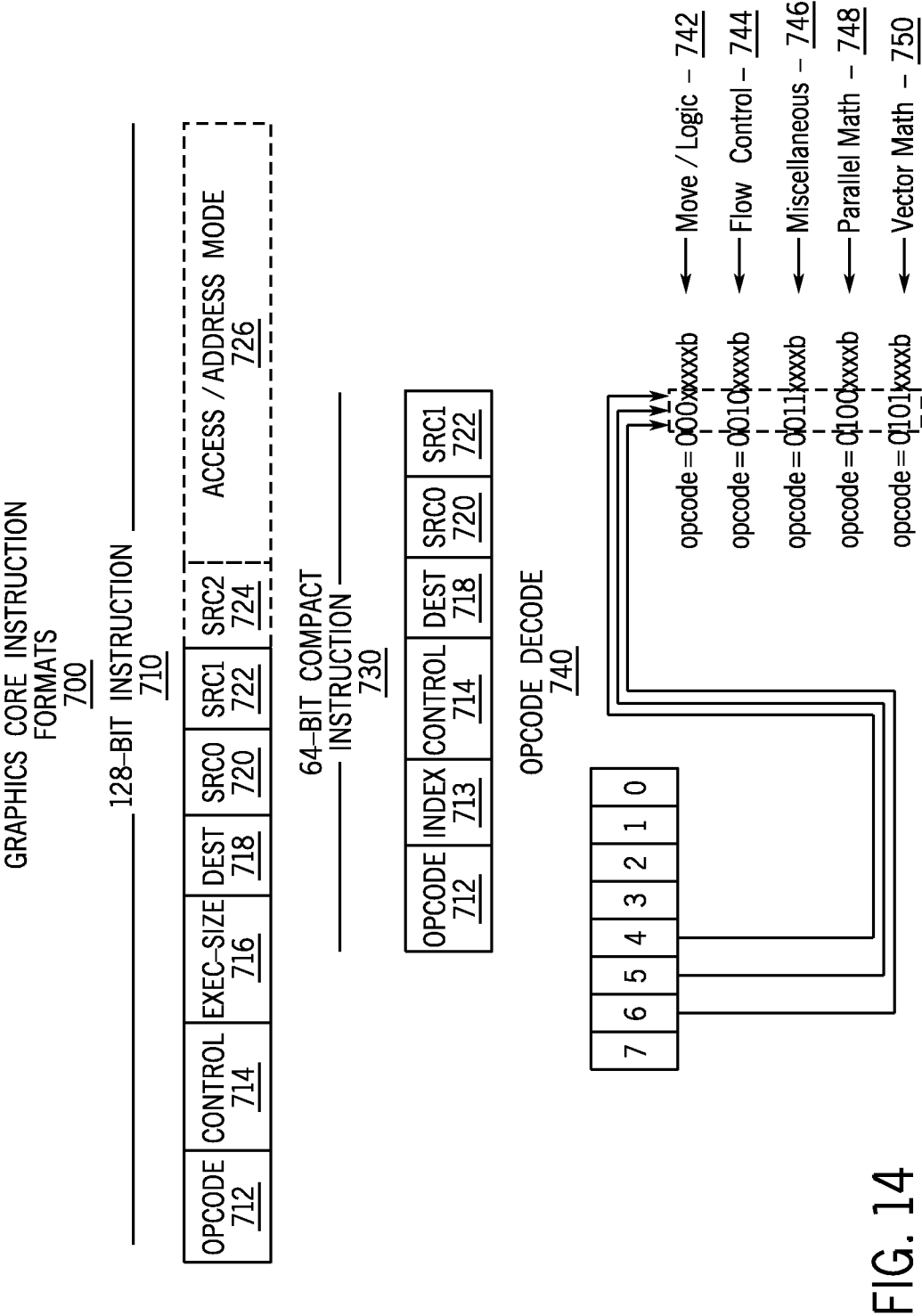


FIG. 13



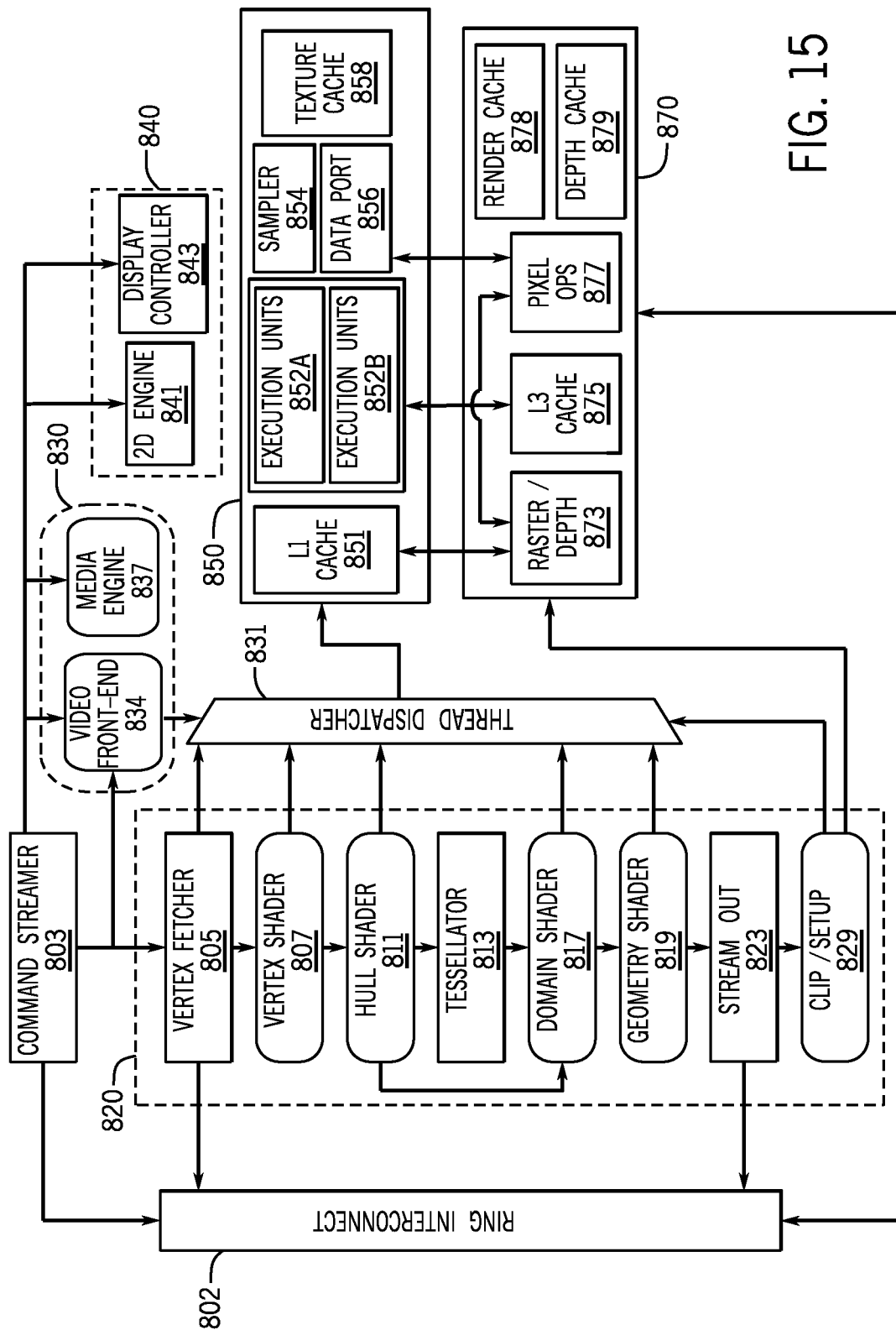


FIG. 15

FIG. 16A

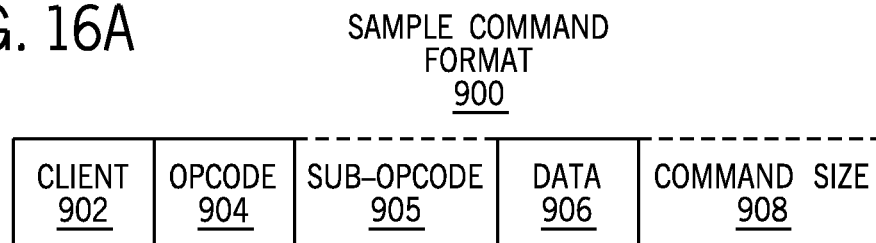
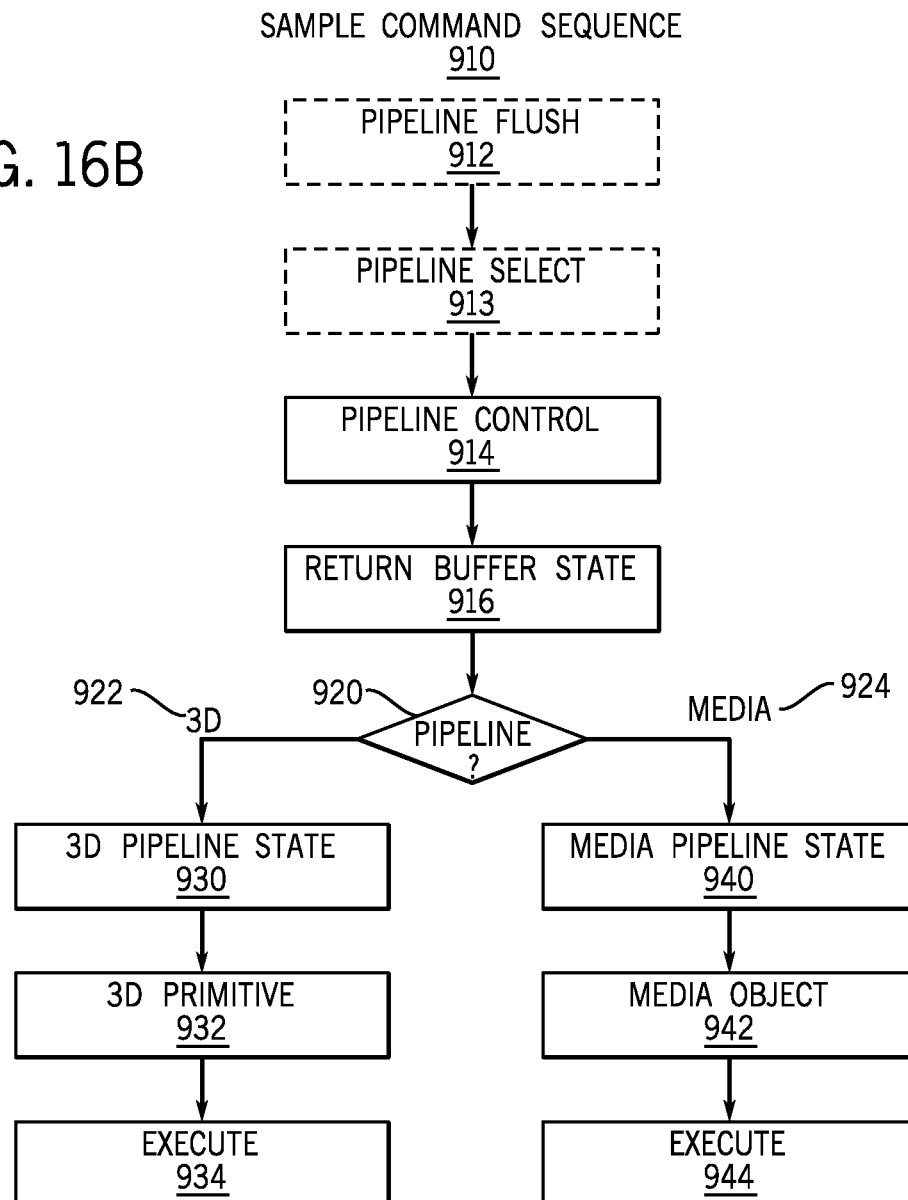


FIG. 16B



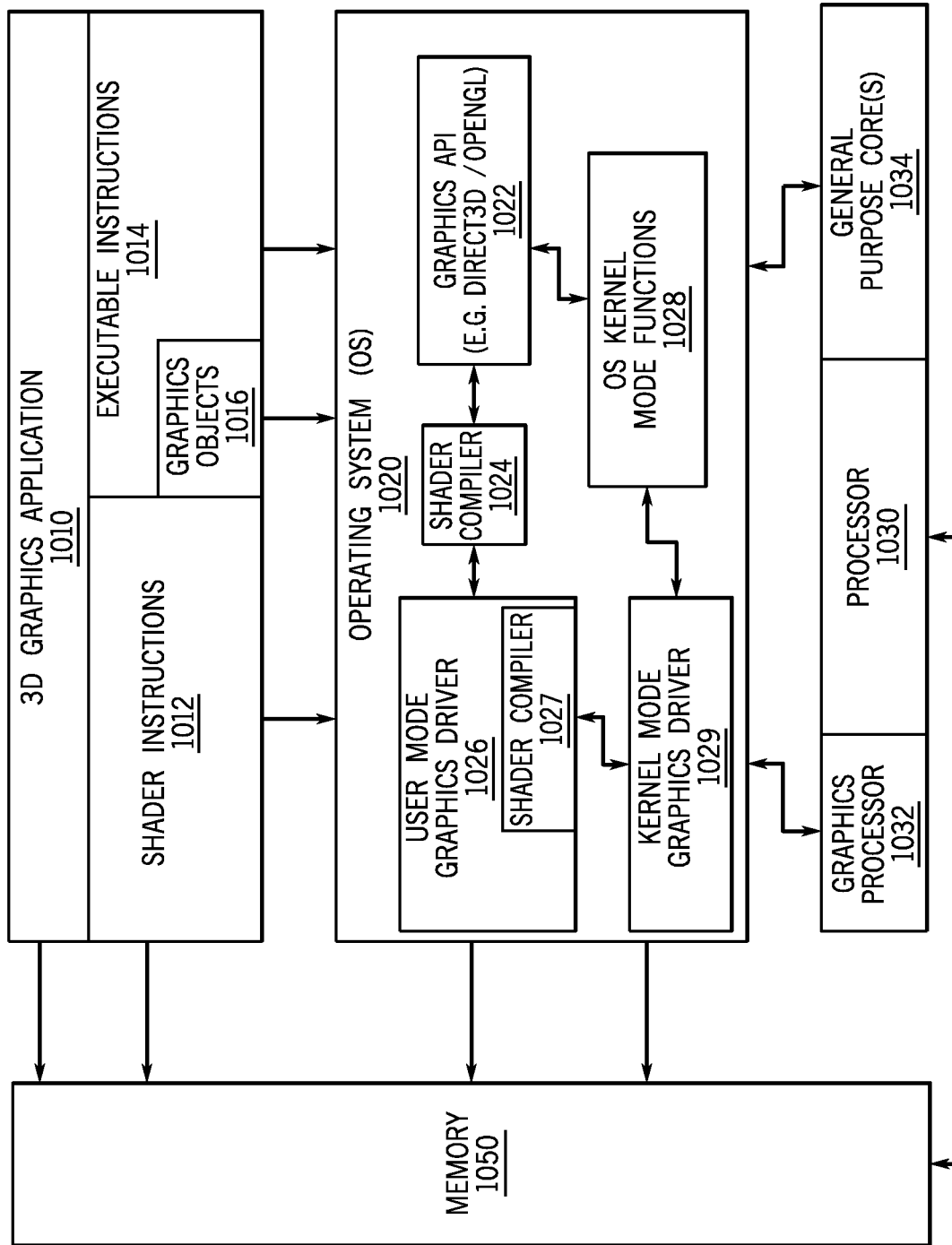


FIG. 17

1

CONSTANT BUFFER SIZE MULTI-SAMPLED ANTI-ALIASING DEPTH COMPRESSION

BACKGROUND

This relates generally to graphics processing.

In graphics processing, the depths of objects within a depiction may be coded. Advantageously, the coding used for the depth information is compressed so that the depth buffer size is reduced, saving costs.

In some cases, a single depth value is used for each pixel. In other cases, called multi-sampled anti-aliasing, multiple depth values are used for each pixel. In this case, there may be some number of samples per pixel and there may be a depth value associated with each of those samples.

The problem that arises with depth compression, for multi-sampled anti-aliasing, is that the size of the depth buffer increases with increasing numbers of samples. Since more samples is generally desirable, and because the cost of a depth buffer increases with increasing size, there is a need for ways to store information with increasing numbers of samples without excessively increasing the depth buffer cost.

BRIEF DESCRIPTION OF THE DRAWINGS

Some embodiments are described with respect to the following figures:

FIG. 1 is a depiction of a depth data storage format for a minimum/maximum scheme according to one embodiment;

FIG. 2 is a depiction of a depth data storage format for a plane equation scheme according to one embodiment;

FIG. 3 is a diagram showing how one triangle may have its depth data stored using two different schemes;

FIG. 4 is a schematic depiction of a depth buffering apparatus according to one embodiment;

FIG. 5 is a depiction of an output rasterization based on conditions in the input range, according to some embodiments;

FIG. 6 is a flow chart for a higher level depth test according to one embodiment;

FIG. 7 is a flow chart for a min/max depth encoding according to one embodiment;

FIG. 8 is a block diagram of a data processing system according to one embodiment;

FIG. 9 is a block diagram of the processor shown in FIG. 8 according to one embodiment;

FIG. 10 is a block diagram of the graphics processor of FIG. 8 according to one embodiment.

FIG. 11 is a block diagram of a graphics processing engine according to one embodiment;

FIG. 12 is a block diagram of a graphics processor according to another embodiment;

FIG. 13 illustrates thread execution logic for one embodiment;

FIG. 14 is a block diagram of a graphics processor execution unit instruction format according to one embodiment;

FIG. 15 is a block diagram of another embodiment of a graphics processor;

FIG. 16A is a block diagram of a graphics processor command format according to one embodiment;

FIG. 16B is a block diagram of a graphics processor command sequence according to one embodiment; and

FIG. 17 is a graphics software architecture for one embodiment.

DETAILED DESCRIPTION

The more samples that are involved, the more depth values that have been traditionally used to process a single pixel.

2

Each sample has an individual depth value that is stored in a depth buffer. By packing the depth data in a way that is independent of the number of samples, so that memory bandwidth is the same regardless of the number of samples, higher numbers of samples per pixel may be used without adversely affecting buffer cost. In some embodiments, the number of pixels per clock in a first level depth test may be increased by operating in the pixel domain, whereas previous solutions operated at the sample level.

The depth values may be stored in one of two ways, while in both cases the size of the depth buffer storage is independent of the number of samples. Specifically, the depth values may be stored either as plane equations or by storing a minimum and maximum value for a group of pixels, each pixel having multiple samples. Both methods of storage (plane equation and minimum and maximum values for a group of pixels) may be used within the same compressed depth buffer, depending on the continuity of the depth data. As used herein, the "continuity" of the depth data indicates whether or not all the pixels and all the samples for a given group or chunk of data are lit. The term "lit" means that all the pixels have a sample with a specified color value and all the samples and all the pixels are actually used in the depicted image. For example in some depictions, a line that is depicted may go through only part of a pixel so that some of the samples associated with the pixel are lit and others are not.

As used herein, portions of a frame including a large number of pixels, are treated as a "group." For example, in one embodiment, a 4x8 group of pixels may be treated as a group which is subdividable into two subgroups of 4x4 pixels. The groups of pixels may be manipulated as a whole. However the size of the group and any subgroup may be different from those mentioned herein.

The depth buffer is divided into groups of pixels, conveniently defined by a closed, rectangular area of pixels, and each of these groups is stored using either the plane equation or the minimum and maximum values of the group, depending on the continuity of the group. These areas may have constant size in the pixel domain and do not change as the number of samples per pixel is increased in one embodiment. The amount of work performed at a first level depth test maps exactly to the size of the stored chunks and is therefore constant in the pixel domain as the number of samples is increased.

The usage model for integrated graphics is shifting towards high end gaming rather than office and business applications. The gaming applications push towards multi-sampling and increased image quality. Enabling increasing numbers of samples per pixel without increasing the memory size or bandwidth may provide more economical solutions in some cases.

In some embodiments a compressed depth buffer stores depth values that are compressed using one of two coding formats. The compressed depth buffer is used for a higher level depth test. In one embodiment, this higher level depth test can test a group of pixels in one clock. The data may be arranged to shadow the uncompressed, regular depth buffer. The higher level depth test sometimes needs to defer to the lower level depth test when it is not possible to return the same test result for all the pixels in the group.

The data in the compressed depth buffer may be organized using a so-called plane bit to indicate its plane in the depth buffer as well as pixel mask information. In the case where the higher level depth test does not return the same result, then the lower level depth test may be used and the data may be stored in uncompressed depth buffer. Generally this is undesirable

because it is more efficient to store the depth values in the compressed depth buffer, if possible.

The first coding format, called the minimum/maximum format, involves storing the minimum and maximum depth values for a group of pixels corresponding to a given region (such as a 4×4 or 8×4 group of pixels). These pixels may each consist of multiple samples. The amount of storage however is not dependent on how many samples are used. The compressed representation, however, is lossy because all of the original per sample data cannot be recovered.

The pixel mask may be part of each entry in the compressed depth buffer. The pixel mask is used to differentiate pixels that are part of a minimum/maximum range from pixels that are not stored using a minimum/maximum approach. The depth of pixels not included in the minimum/maximum range is considered to be equal to the depth buffer clear value. The depth buffer clear value is a starting or default value for all pixels in the buffer at the beginning of a rendering process. The depth values in the depth buffer are cleared to this clear value at the start of each new frame. The clear value can be any value in the range 0 to 1. It is stored “on the side” as one floating point value global to the entire z buffer.

The pixel mask is set whenever any pixel with at least one sample is lit. When there are partially lit pixels, (i.e. pixels with not all samples lit), drawn on top of the clear buffer, a clear value is included in the minimum and maximum range. When partially lit pixels are drawn on top of previously rendered parts of the image, the old ranges are incorporated into the new range and written back to the compressed depth buffer.

A partially lit pixel mask takes the logical OR of the samples in the group and the fully lit pixel mask uses the logical AND of the samples in one embodiment.

Thus, for the groups that are represented by their minimums and maximums, there are basically six cases in one embodiment. When the original or old hierarchical depth (HZ) mask is read from the compressed depth buffer is all ones, and the partially lit source mask for the new data are all ones, then the source (i.e. new frames) range replaces the old range in one embodiment. If both sub-groups, (such as 4×4 areas) of pixels are fully lit, then the group of pixels can be represented by its plane equation.

The next case is when the old HZ mask is read from the compressed buffer is all ones for the group but the fully lit mask has at least one zero. This would mean that at least one pixel has at least one sample which is not lit. In this case, the new range combines with the old range.

The next case is again when the old mask read from the compressed buffer is all ones and both the fully lit and the partially lit source mask has at least one pixel which is zero. In this case the source range combines with the old range.

The next three cases are all cases in which the old HZ mask read from the compressed buffer is all zeroes. In the first of these three cases, both the fully lit and the partially lit source masks are all ones. Then the source range replaces the clear values. If both 4×4 groups of pixels are fully lit, then the group of pixels can be represented by its plane equation.

The next instance for the old mask having all zeroes is where the fully lit source mask has at least one pixel that is zero and the partially lit source mask has all ones. In this case the source range combines with the clear value.

The final and third instance where the original mask is all zeroes is the case where the fully lit source mask and the partially lit source mask have zeroes. In this case the source range replaces the old range.

The plane equation may be used to represent a group of pixels when the entire group of pixels is lit. Then, the exact

depth data for all the samples can be found by evaluating the plane equation where the depth value Z is equal to $Z = x * CX + y * CY$ where CX is the x coordinate of the depth value and CY is its y coordinate. The xy coordinates can be fractional giving an exact depth of the samples within each pixel.

This plane equation also uses a fixed amount of storage that is not dependent on the number of samples. It is written back to the compressed buffer whenever the entire group of pixels passes the depth test. All samples for all pixels are lit and no further pixel test in the pipeline can discard any pixels.

Both formats include the plane bit in the same most significant bit position in one embodiment. The plane bit tells the hardware how to interpret the data that follows. The higher level depth test is more accurate whenever the input is in the plane format. The higher level depth test sometimes needs to fall back on the lower level depth test when the input format is in the minimum and maximum format.

Referring to FIG. 1, a group composed a 8×4 chunk of data made up of two 4×4 chunks of data is depicted. In this example the minimum/maximum format is used when the depth data is not continuous and so it is not representable by a single plane equation. In this case, the minimum/maximum format takes up 64 bits of data regardless of the number of samples. The first 16 bits are for the pixel mask, the next 24 bits may be used for the maximum depth value, the next 23 bits may be used for the minimum depth value and the last bit may be the plane bit for each 4×4 chunk in one embodiment.

In contrast as shown in FIG. 2, an 8×4 chunk of pixels is stored in the plane equation format which takes up 128 bits of storage in this example. This corresponds to the same amount of data as two minimum/maximum format slots of 4×4 size. In this case, the first 32 bits are for the CY coefficient, the next 32 bits are for the CX coefficient, the next 31 bits are for the CO coefficient and the last bit is for the plane bit. Other organizations of the bits may also be used. The plane format is used when the data is continuous and occupies the same space as two minimum/maximum format 4×4 slots.

FIG. 3 shows an example of a rasterization of a triangle using two different storage formats. The letter m represents the minimum/maximum format and the word “plane” indicates the use of the plane equation to code the data.

The storage requirements of the compressed depth buffer remain constant with respect to the amount of samples used to render each pixel.

Referring to FIG. 4, a depth buffering scheme may receive data for depth buffer testing from a rasterizer 10. First the data from the rasterizer is subjected to the higher level depth test at 12. Data may be stored in compressed format in the compressed depth buffer 14. If the higher level depth test does not work, then the lower level depth test 16 must be used and the data stored in a regular uncompressed depth buffer 18.

FIG. 5 is a depiction of the strategies for a depth buffering based on the old HZ mask read from the compressed depth buffer (left column) in a case where there is a fully lit source mask (second column), and a partially lit source mask (third column). The output range action in the cases when no other pixel is enabled are depicted in the rightmost column.

FIGS. 6 and 7 depicts sequences which may be implemented in software, firmware and/or hardware. In software and firmware embodiments the sequences may be implemented by computer executable instructions stored in one or more non-transitory computer readable media such as magnetic, optical or semiconductor storages.

The sequence of FIG. 6 may be used for the higher level depth test. The depth values are received at block 20 for a group of pixels. Each pixel consisting of multiple samples. A check at diamond 22 determines whether the data is continu-

5

ous. If so, the data may be represented as a plane equation and stored as indicated in block 24 in the compressed depth buffer. Otherwise the data may be stored as two minimum/maximum chunks as indicated in block 26.

Referring to FIG. 7, the sequence for the minimum/maximum storage technique is depicted according to one embodiment. Initially a check at diamond 30 determines whether the source mask is not lit. If so, the old minimum and maximum range is maintained as indicated in block 32.

If the source mask is not lit then a check at diamond 34 determines whether the source mask is fully lit. If so, the old range is replaced with the source's minimum and maximum range as indicated in block 36. If not, a check at diamond 38 determines whether the source mask has partially lit pixels.

If so, a check at diamond 40 determines whether any partial pixel overlaps with zero in a destination mask. If not, the old range is combined with the source range as indicated in block 42.

If the check at diamond 38 determines that the source mask does not have partially lit pixels, then a check at diamond 44 determines whether any unlit source pixel overlaps with a lit destination pixel. If so, the flow goes to block 42 combining the old range with the source range. Otherwise, the old range is replaced with the source minimum maximum range in block 36.

If the check at diamond 40 determines that any pixel does overlap with zero in the destination mask then a check at diamond 46 determines whether any partial pixel overlaps with a lit destination pixel. If so, the old range clear value and source range are combined at block 48. Otherwise, the old range is replaced with the combination of the source range and the clear range at block 50.

Then in any case a new HZ mask is a logical "OR" of the old mask and the source mask. The masks are one bit per pixel. In an multi-sampled, anti-aliasing mode, partially lit pixels are treated as lit as indicated in block 52.

FIG. 8 is a block diagram of a data processing system 100, according to an embodiment. The data processing system 100 includes one or more processors 102 and one or more graphics processors 108, and may be a single processor desktop system, a multiprocessor workstation system, or a server system having a large number of processors 102 or processor cores 107. In one embodiment, the data processing system 100 is a system on a chip integrated circuit (SOC) for use in mobile, handheld, or embedded devices.

An embodiment of the data processing system 100 can include, or be incorporated within a server-based gaming platform, a game console, including a game and media console, a mobile gaming console, a handheld game console, or an online game console. In one embodiment, the data processing system 100 is a mobile phone, smart phone, tablet computing device or mobile Internet device. The data processing system 100 can also include, couple with, or be integrated within a wearable device, such as a smart watch wearable device, smart eyewear device, augmented reality device, or virtual reality device. In one embodiment, the data processing system 100 is a television or set top box device having one or more processors 102 and a graphical interface generated by one or more graphics processors 108.

The one or more processors 102 each include one or more processor cores 107 to process instructions which, when executed, perform operations for system and user software. In one embodiment, each of the one or more processor cores 107 is configured to process a specific instruction set 109. The instruction set 109 may facilitate complex instruction set computing (CISC), reduced instruction set computing (RISC), or computing via a very long instruction word

6

(VLIW). Multiple processor cores 107 may each process a different instruction set 109 which may include instructions to facilitate the emulation of other instruction sets. A processor core 107 may also include other processing devices, such as a digital signal processor (DSP).

In one embodiment, the processor 102 includes cache memory 104. Depending on the architecture, the processor 102 can have a single internal cache or multiple levels of internal cache. In one embodiment, the cache memory is shared among various components of the processor 102. In one embodiment, the processor 102 also uses an external cache (e.g., a Level 3 (L3) cache or last level cache (LLC)) (not shown) which may be shared among the processor cores 107 using known cache coherency techniques. A register file 106 is additionally included in the processor 102 which may include different types of registers for storing different types of data (e.g., integer registers, floating point registers, status registers, and an instruction pointer register). Some registers may be general-purpose registers, while other registers may be specific to the design of the processor 102.

The processor 102 is coupled to a processor bus 110 to transmit data signals between the processor 102 and other components in the system 100. The system 100 uses an exemplary 'hub' system architecture, including a memory controller hub 116 and an input output (I/O) controller hub 130. The memory controller hub 116 facilitates communication between a memory device and other components of the system 100, while the I/O controller hub (ICH) 130 provides connections to I/O devices via a local I/O bus.

The memory device 120, can be a dynamic random access memory (DRAM) device, a static random access memory (SRAM) device, flash memory device, or some other memory device having suitable performance to serve as process memory. The memory 120 can store data 122 and instructions 121 for use when the processor 102 executes a process. The memory controller hub 116 also couples with an optional external graphics processor 112, which may communicate with the one or more graphics processors 108 in the processors 102 to perform graphics and media operations.

The ICH 130 enables peripherals to connect to the memory 120 and processor 102 via a high-speed I/O bus. The I/O peripherals include an audio controller 146, a firmware interface 128, a wireless transceiver 126 (e.g., Wi-Fi, Bluetooth), a data storage device 124 (e.g., hard disk drive, flash memory, etc.), and a legacy I/O controller for coupling legacy (e.g., Personal System 2 (PS/2)) devices to the system. One or more Universal Serial Bus (USB) controllers 142 connect input devices, such as keyboard and mouse 144 combinations. A network controller 134 may also couple to the ICH 130. In one embodiment, a high-performance network controller (not shown) couples to the processor bus 110.

FIG. 9 is a block diagram of an embodiment of a processor 200 having one or more processor cores 202A-N, an integrated memory controller 214, and an integrated graphics processor 208. The processor 200 can include additional cores up to and including additional core 202N represented by the dashed lined boxes. Each of the cores 202A-N includes one or more internal cache units 204A-N. In one embodiment each core also has access to one or more shared cached units 206.

The internal cache units 204A-N and shared cache units 206 represent a cache memory hierarchy within the processor 200. The cache memory hierarchy may include at least one level of instruction and data cache within each core and one or more levels of shared mid-level cache, such as a level 2 (L2), level 3 (L3), level 4 (L4), or other levels of cache, where the highest level of cache before external memory is classified as

the last level cache (LLC). In one embodiment, cache coherency logic maintains coherency between the various cache units **206** and **204A-N**.

The processor **200** may also include a set of one or more bus controller units **216** and a system agent **210**. The one or more bus controller units manage a set of peripheral buses, such as one or more Peripheral Component Interconnect buses (e.g., PCI, PCI Express). The system agent **210** provides management functionality for the various processor components. In one embodiment, the system agent **210** includes one or more integrated memory controllers **214** to manage access to various external memory devices (not shown).

In one embodiment, one or more of the cores **202A-N** include support for simultaneous multi-threading. In such embodiment, the system agent **210** includes components for coordinating and operating cores **202A-N** during multi-threaded processing. The system agent **210** may additionally include a power control unit (PCU), which includes logic and components to regulate the power state of the cores **202A-N** and the graphics processor **208**.

The processor **200** additionally includes a graphics processor **208** to execute graphics processing operations. In one embodiment, the graphics processor **208** couples with the set of shared cache units **206**, and the system agent unit **210**, including the one or more integrated memory controllers **214**. In one embodiment, a display controller **211** is coupled with the graphics processor **208** to drive graphics processor output to one or more coupled displays. The display controller **211** may be separate module coupled with the graphics processor via at least one interconnect, or may be integrated within the graphics processor **208** or system agent **210**.

In one embodiment a ring based interconnect unit **212** is used to couple the internal components of the processor **200**, however an alternative interconnect unit may be used, such as a point to point interconnect, a switched interconnect, or other techniques, including techniques well known in the art. In one embodiment, the graphics processor **208** couples with the ring interconnect **212** via an I/O link **213**.

The exemplary I/O link **213** represents at least one of multiple varieties of I/O interconnects, including an on package I/O interconnect which facilitates communication between various processor components and a high-performance embedded memory module **218**, such as an eDRAM module. In one embodiment each of the cores **202-N** and the graphics processor **208** use the embedded memory modules **218** as shared last level cache.

In one embodiment cores **202A-N** are homogenous cores executing the same instruction set architecture. In another embodiment, the cores **202A-N** are heterogeneous in terms of instruction set architecture (ISA), where one or more of the cores **202A-N** execute a first instruction set, while at least one of the other cores executes a subset of the first instruction set or a different instruction set.

The processor **200** can be a part of or implemented on one or more substrates using any of a number of process technologies, for example, Complementary metal-oxide-semiconductor (CMOS), Bipolar Junction/Complementary metal-oxide-semiconductor (BiCMOS) or N-type metal-oxide-semiconductor logic (NMOS). Additionally, the processor **200** can be implemented on one or more chips or as a system on a chip (SOC) integrated circuit having the illustrated components, in addition to other components.

FIG. 10 is a block diagram of one embodiment of a graphics processor **300** which may be a discrete graphics processing unit, or may be graphics processor integrated with a plurality of processing cores. In one embodiment, the graphics

processor is communicated with via a memory mapped I/O interface to registers on the graphics processor and via commands placed into the processor memory. The graphics processor **300** includes a memory interface **314** to access memory. The memory interface **314** can be an interface to local memory, one or more internal caches, one or more shared external caches, and/or to system memory.

The graphics processor **300** also includes a display controller **302** to drive display output data to a display device **320**. The display controller **302** includes hardware for one or more overlay planes for the display and composition of multiple layers of video or user interface elements. In one embodiment the graphics processor **300** includes a video codec engine **306** to encode, decode, or transcode media to, from, or between one or more media encoding formats, including, but not limited to Moving Picture Experts Group (MPEG) formats such as MPEG-2, Advanced Video Coding (AVC) formats such as H.264/MPEG-4 AVC, as well as the Society of Motion Picture & Television Engineers (SMPTE) 421M/VC-1, and Joint Photographic Experts Group (JPEG) formats such as JPEG, and Motion JPEG (MJPEG) formats.

In one embodiment, the graphics processor **300** includes a block image transfer (BLIT) engine **304** to perform two-dimensional (2D) rasterizer operations including, for example, bit-boundary block transfers. However, in one embodiment, 2D graphics operations are performed using one or more components of the graphics-processing engine (GPE) **310**. The graphics-processing engine **310** is a compute engine for performing graphics operations, including three-dimensional (3D) graphics operations and media operations.

The GPE **310** includes a 3D pipeline **312** for performing 3D operations, such as rendering three-dimensional images and scenes using processing functions that act upon 3D primitive shapes (e.g., rectangle, triangle, etc.). The 3D pipeline **312** includes programmable and fixed function elements that perform various tasks within the element and/or spawn execution threads to a 3D/Media sub-system **315**. While the 3D pipeline **312** can be used to perform media operations, an embodiment of the GPE **310** also includes a media pipeline **316** that is specifically used to perform media operations, such as video post processing and image enhancement.

In one embodiment, the media pipeline **316** includes fixed function or programmable logic units to perform one or more specialized media operations, such as video decode acceleration, video de-interlacing, and video encode acceleration in place of, or on behalf of the video codec engine **306**. In one embodiment, the media pipeline **316** additionally includes a thread spawning unit to spawn threads for execution on the 3D/Media sub-system **315**. The spawned threads perform computations for the media operations on one or more graphics execution units included in the 3D/Media sub-system.

The 3D/Media subsystem **315** includes logic for executing threads spawned by the 3D pipeline **312** and media pipeline **316**. In one embodiment, the pipelines send thread execution requests to the 3D/Media subsystem **315**, which includes thread dispatch logic for arbitrating and dispatching the various requests to available thread execution resources. The execution resources include an array of graphics execution units to process the 3D and media threads. In one embodiment, the 3D/Media subsystem **315** includes one or more internal caches for thread instructions and data. In one embodiment, the subsystem also includes shared memory, including registers and addressable memory, to share data between threads and to store output data.

FIG. 11 is a block diagram of an embodiment of a graphics processing engine **410** for a graphics processor. In one embodiment, the graphics processing engine (GPE) **410** is a

version of the GPE 310 shown in FIG. 10. The GPE 410 includes a 3D pipeline 412 and a media pipeline 416, each of which can be either different from or similar to the implementations of the 3D pipeline 312 and the media pipeline 316 of FIG. 10.

In one embodiment, the GPE 410 couples with a command streamer 403, which provides a command stream to the GPE 3D and media pipelines 412, 416. The command streamer 403 is coupled to memory, which can be system memory, or one or more of internal cache memory and shared cache memory. The command streamer 403 receives commands from the memory and sends the commands to the 3D pipeline 412 and/or media pipeline 416. The 3D and media pipelines process the commands by performing operations via logic within the respective pipelines or by dispatching one or more execution threads to the execution unit array 414. In one embodiment, the execution unit array 414 is scalable, such that the array includes a variable number of execution units based on the target power and performance level of the GPE 410.

A sampling engine 430 couples with memory (e.g., cache memory or system memory) and the execution unit array 414. In one embodiment, the sampling engine 430 provides a memory access mechanism for the scalable execution unit array 414 that allows the execution array 414 to read graphics and media data from memory. In one embodiment, the sampling engine 430 includes logic to perform specialized image sampling operations for media.

The specialized media sampling logic in the sampling engine 430 includes a de-noise/de-interlace module 432, a motion estimation module 434, and an image scaling and filtering module 436. The de-noise/de-interlace module 432 includes logic to perform one or more of a de-noise or a de-interlace algorithm on decoded video data. The de-interlace logic combines alternating fields of interlaced video content into a single frame of video. The de-noise logic reduces or remove data noise from video and image data. In one embodiment, the de-noise logic and de-interlace logic are motion adaptive and use spatial or temporal filtering based on the amount of motion detected in the video data. In one embodiment, the de-noise/de-interlace module 432 includes dedicated motion detection logic (e.g., within the motion estimation engine 434).

The motion estimation engine 434 provides hardware acceleration for video operations by performing video acceleration functions such as motion vector estimation and prediction on video data. The motion estimation engine determines motion vectors that describe the transformation of image data between successive video frames. In one embodiment, a graphics processor media codec uses the video motion estimation engine 434 to perform operations on video at the macro-block level that may otherwise be computationally intensive to perform using a general-purpose processor. In one embodiment, the motion estimation engine 434 is generally available to graphics processor components to assist with video decode and processing functions that are sensitive or adaptive to the direction or magnitude of the motion within video data.

The image scaling and filtering module 436 performs image-processing operations to enhance the visual quality of generated images and video. In one embodiment, the scaling and filtering module 436 processes image and video data during the sampling operation before providing the data to the execution unit array 414.

In one embodiment, the graphics processing engine 410 includes a data port 444, which provides an additional mechanism for graphics subsystems to access memory. The data port 444 facilitates memory access for operations including

render target writes, constant buffer reads, scratch memory space reads/writes, and media surface accesses. In one embodiment, the data port 444 includes cache memory space to cache accesses to memory. The cache memory can be a single data cache or separated into multiple caches for the multiple subsystems that access memory via the data port (e.g., a render buffer cache, a constant buffer cache, etc.). In one embodiment, threads executing on an execution unit in the execution unit array 414 communicate with the data port by exchanging messages via a data distribution interconnect that couples each of the sub-systems of the graphics processing engine 410.

FIG. 12 is a block diagram of another embodiment of a graphics processor. In one embodiment, the graphics processor includes a ring interconnect 502, a pipeline front-end 504, a media engine 537, and graphics cores 580A-N. The ring interconnect 502 couples the graphics processor to other processing units, including other graphics processors or one or more general-purpose processor cores. In one embodiment, the graphics processor is one of many processors integrated within a multi-core processing system.

The graphics processor receives batches of commands via the ring interconnect 502. The incoming commands are interpreted by a command streamer 503 in the pipeline front-end 504. The graphics processor includes scalable execution logic to perform 3D geometry processing and media processing via the graphics core(s) 580A-N. For 3D geometry processing commands, the command streamer 503 supplies the commands to the geometry pipeline 536. For at least some media processing commands, the command streamer 503 supplies the commands to a video front end 534, which couples with a media engine 537. The media engine 537 includes a video quality engine (VQE) 530 for video and image post processing and a multi-format encode/decode (MFX) 533 engine to provide hardware-accelerated media data encode and decode. The geometry pipeline 536 and media engine 537 each generate execution threads for the thread execution resources provided by at least one graphics core 580A.

The graphics processor includes scalable thread execution resources featuring modular cores 580A-N (sometimes referred to as core slices), each having multiple sub-cores 550A-N, 560A-N (sometimes referred to as core sub-slices). The graphics processor can have any number of graphics cores 580A through 580N. In one embodiment, the graphics processor includes a graphics core 580A having at least a first sub-core 550A and a second core sub-core 560A. In another embodiment, the graphics processor is a low power processor with a single sub-core (e.g., 550A). In one embodiment, the graphics processor includes multiple graphics cores 580A-N, each including a set of first sub-cores 550A-N and a set of second sub-cores 560A-N. Each sub-core in the set of first sub-cores 550A-N includes at least a first set of execution units 552A-N and media/texture samplers 554A-N. Each sub-core in the set of second sub-cores 560A-N includes at least a second set of execution units 562A-N and samplers 564A-N. In one embodiment, each sub-core 550A-N, 560A-N shares a set of shared resources 570A-N. In one embodiment, the shared resources include shared cache memory and pixel operation logic. Other shared resources may also be included in the various embodiments of the graphics processor.

FIG. 13 illustrates thread execution logic 600 including an array of processing elements employed in one embodiment of a graphics processing engine. In one embodiment, the thread execution logic 600 includes a pixel shader 602, a thread dispatcher 604, instruction cache 606, a scalable execution unit array including a plurality of execution units 608A-N, a sampler 610, a data cache 612, and a data port 614. In one

embodiment the included components are interconnected via an interconnect fabric that links to each of the components. The thread execution logic **600** includes one or more connections to memory, such as system memory or cache memory, through one or more of the instruction cache **606**, the data port **614**, the sampler **610**, and the execution unit array **608A-N**. In one embodiment, each execution unit (e.g. **608A**) is an individual vector processor capable of executing multiple simultaneous threads and processing multiple data elements in parallel for each thread. The execution unit array **608A-N** includes any number individual execution units.

In one embodiment, the execution unit array **608A-N** is primarily used to execute “shader” programs. In one embodiment, the execution units in the array **608A-N** execute an instruction set that includes native support for many standard 3D graphics shader instructions, such that shader programs from graphics libraries (e.g., Direct 3D and OpenGL) are executed with a minimal translation. The execution units support vertex and geometry processing (e.g., vertex programs, geometry programs, vertex shaders), pixel processing (e.g., pixel shaders, fragment shaders) and general-purpose processing (e.g., compute and media shaders).

Each execution unit in the execution unit array **608A-N** operates on arrays of data elements. The number of data elements is the “execution size,” or the number of channels for the instruction. An execution channel is a logical unit of execution for data element access, masking, and flow control within instructions. The number of channels may be independent of the number of physical ALUs or FPUs for a particular graphics processor. The execution units **608A-N** support integer and floating-point data types.

The execution unit instruction set includes single instruction multiple data (SIMD) instructions. The various data elements can be stored as a packed data type in a register and the execution unit will process the various elements based on the data size of the elements. For example, when operating on a 256-bit wide vector, the 256 bits of the vector are stored in a register and the execution unit operates on the vector as four separate 64-bit packed data elements (quad-word (QW) size data elements), eight separate 32-bit packed data elements (double word (DW) size data elements), sixteen separate 16-bit packed data elements (word (W) size data elements), or thirty-two separate 8-bit data elements (byte (B) size data elements). However, different vector widths and register sizes are possible.

One or more internal instruction caches (e.g., **606**) are included in the thread execution logic **600** to cache thread instructions for the execution units. In one embodiment, one or more data caches (e.g., **612**) are included to cache thread data during thread execution. A sampler **610** is included to provide texture sampling for 3D operations and media sampling for media operations. In one embodiment, the sampler **610** includes specialized texture or media sampling functionality to process texture or media data during the sampling process before providing the sampled data to an execution unit.

During execution, the graphics and media pipelines send thread initiation requests to the thread execution logic **600** via thread spawning and dispatch logic. The thread execution logic **600** includes a local thread dispatcher **604** that arbitrates thread initiation requests from the graphics and media pipelines and instantiates the requested threads on one or more execution units **608A-N**. For example, the geometry pipeline (e.g., **536** of FIG. **8**) dispatches vertex processing, tessellation, or geometry processing threads to the thread execution

logic **600**. The thread dispatcher **604** can also process runtime thread spawning requests from the executing shader programs.

Once a group of geometric objects have been processed and rasterized into pixel data, the pixel shader **602** is invoked to further compute output information and cause results to be written to output surfaces (e.g., color buffers, depth buffers, stencil buffers, etc.). In one embodiment, the pixel shader **602** calculates the values of the various vertex attributes that are to be interpolated across the rasterized object. The pixel shader **602** then executes an API-supplied pixel shader program. To execute the pixel shader program, the pixel shader **602** dispatches threads to an execution unit (e.g., **608A**) via the thread dispatcher **604**. The pixel shader **602** uses texture sampling logic in the sampler **610** to access texture data in texture maps stored in memory. Arithmetic operations on the texture data and the input geometry data compute pixel color data for each geometric fragment, or discards one or more pixels from further processing.

In one embodiment, the data port **614** provides a memory access mechanism for the thread execution logic **600** output processed data to memory for processing on a graphics processor output pipeline. In one embodiment, the data port **614** includes or couples to one or more cache memories (e.g., data cache **612**) to cache data for memory access via the data port.

FIG. **14** is a block diagram illustrating a graphics processor execution unit instruction format according to an embodiment. In one embodiment, the graphics processor execution units support an instruction set having instructions in multiple formats. The solid lined boxes illustrate the components that are generally included in an execution unit instruction, while the dashed lines include components that are optional or that are only included in a sub-set of the instructions. The instruction format described and illustrated are macro-instructions, in that they are instructions supplied to the execution unit, as opposed to micro-operations resulting from instruction decode once the instruction is processed.

In one embodiment, the graphics processor execution units natively support instructions in a 128-bit format **710**. A 64-bit compacted instruction format **730** is available for some instructions based on the selected instruction, instruction options, and number of operands. The native 128-bit format **710** provides access to all instruction options, while some options and operations are restricted in the 64-bit format **730**. The native instructions available in the 64-bit format **730** varies by embodiment. In one embodiment, the instruction is compacted in part using a set of index values in an index field **713**. The execution unit hardware references a set of compaction tables based on the index values and uses the compaction table outputs to reconstruct a native instruction in the 128-bit format **710**.

For each format, an instruction opcode **712** defines the operation that the execution unit is to perform. The execution units execute each instruction in parallel across the multiple data elements of each operand. For example, in response to an add instruction the execution unit performs a simultaneous add operation across each color channel representing a texture element or picture element. By default, the execution unit performs each instruction across all data channels of the operands. An instruction control field **712** enables control over certain execution options, such as channels selection (e.g., predication) and data channel order (e.g., swizzle). For 128-bit instructions **710** an exec-size field **716** limits the number of data channels that will be executed in parallel. The exec-size field **716** is not available for use in the 64-bit compact instruction format **730**.

13

Some execution unit instructions have up to three operands including two source operands, **src0 720**, **src1 722**, and one destination **718**. In one embodiment, the execution units support dual destination instructions, where one of the destinations is implied. Data manipulation instructions can have a third source operand (e.g., **SRC2 724**), where the instruction opcode **JJ12** determines the number of source operands. An instruction's last source operand can be an immediate (e.g., hard-coded) value passed with the instruction.

In one embodiment instructions are grouped based on opcode bit-fields to simplify Opcode decode **740**. For an 8-bit opcode, bits **4**, **5**, and **6** allow the execution unit to determine the type of opcode. The precise opcode grouping shown is exemplary. In one embodiment, a move and logic opcode group **742** includes data movement and logic instructions (e.g., **mov**, **cmp**). The move and logic group **742** shares the five most significant bits (MSB), where move instructions are in the form of **0000xxxxb** (e.g., **0x0x**) and logic instructions are in the form of **0001xxxxb** (e.g., **0x01**). A flow control instruction group **744** (e.g., **call**, **jmp**) includes instructions in the form of **0010xxxxb** (e.g., **0x20**). A miscellaneous instruction group **746** includes a mix of instructions, including synchronization instructions (e.g., **wait**, **send**) in the form of **0011xxxxb** (e.g., **0x30**). A parallel math instruction group **748** includes component-wise arithmetic instructions (e.g., **add**, **mul**) in the form of **0100xxxxb** (e.g., **0x40**). The parallel math group **748** performs the arithmetic operations in parallel across data channels. The vector math group **750** includes arithmetic instructions (e.g., **dp4**) in the form of **0101xxxxb** (e.g., **0x50**). The vector math group performs arithmetic such as dot product calculations on vector operands.

FIG. 15 is a block diagram of another embodiment of a graphics processor which includes a graphics pipeline **820**, a media pipeline **830**, a display engine **840**, thread execution logic **850**, and a render output pipeline **870**. In one embodiment, the graphics processor is a graphics processor within a multi-core processing system that includes one or more general purpose processing cores. The graphics processor is controlled by register writes to one or more control registers (not shown) or via commands issued to the graphics processor via a ring interconnect **802**. The ring interconnect **802** couples the graphics processor to other processing components, such as other graphics processors or general-purpose processors. Commands from the ring interconnect are interpreted by a command streamer **803** which supplies instructions to individual components of the graphics pipeline **820** or media pipeline **830**.

The command streamer **803** directs the operation of a vertex fetcher **805** component that reads vertex data from memory and executes vertex-processing commands provided by the command streamer **803**. The vertex fetcher **805** provides vertex data to a vertex shader **807**, which performs coordinate space transformation and lighting operations to each vertex. The vertex fetcher **805** and vertex shader **807** execute vertex-processing instructions by dispatching execution threads to the execution units **852A**, **852B** via a thread dispatcher **831**.

In one embodiment, the execution units **852A**, **852B** are an array of vector processors having an instruction set for performing graphics and media operations. The execution units **852A**, **852B** have an attached L1 cache **851** that is specific for each array or shared between the arrays. The cache can be configured as a data cache, an instruction cache, or a single cache that is partitioned to contain data and instructions in different partitions.

In one embodiment, the graphics pipeline **820** includes tessellation components to perform hardware-accelerated

14

tessellation of 3D objects. A programmable hull shader **811** configures the tessellation operations. A programmable domain shader **817** provides back-end evaluation of tessellation output. A tessellator **813** operates at the direction of the hull shader **811** and contains special purpose logic to generate a set of detailed geometric objects based on a coarse geometric model that is provided as input to the graphics pipeline **820**. If tessellation is not used, the tessellation components **811**, **813**, **817** can be bypassed.

The complete geometric objects can be processed by a geometry shader **819** via one or more threads dispatched to the execution units **852A**, **852B**, or can proceed directly to the clipper **829**. The geometry shader operates on entire geometric objects, rather than vertices or patches of vertices as in previous stages of the graphics pipeline. If the tessellation is disabled the geometry shader **819** receives input from the vertex shader **807**. The geometry shader **819** is programmable by a geometry shader program to perform geometry tessellation if the tessellation units are disabled.

Prior to rasterization, vertex data is processed by a clipper **829**, which is either a fixed function clipper or a programmable clipper having clipping and geometry shader functions. In one embodiment, a rasterizer **873** in the render output pipeline **870** dispatches pixel shaders to convert the geometric objects into their per pixel representations. In one embodiment, pixel shader logic is included in the thread execution logic **850**.

The graphics engine has an interconnect bus, interconnect fabric, or some other interconnect mechanism that allows data and message passing amongst the major components of the graphics engine. In one embodiment the execution units **852A**, **852B** and associated cache(s) **851**, texture and media sampler **854**, and texture/sampler cache **858** interconnect via a data port **856** to perform memory access and communicate with render output pipeline components of the graphics engine. In one embodiment, the sampler **854**, caches **851**, **858** and execution units **852A**, **852B** each have separate memory access paths.

In one embodiment, the render output pipeline **870** contains a rasterizer and depth test component **873** that converts vertex-based objects into their associated pixel-based representation. In one embodiment, the rasterizer logic includes a windower/masker unit to perform fixed function triangle and line rasterization. An associated render and depth buffer caches **878**, **879** are also available in one embodiment. A pixel operations component **877** performs pixel-based operations on the data, though in some instances, pixel operations associated with 2D operations (e.g. bit block image transfers with blending) are performed by the 2D engine **841**, or substituted at display time by the display controller **843** using overlay display planes. In one embodiment a shared L3 cache **875** is available to all graphics components, allowing the sharing of data without the use of main system memory.

The graphics processor media pipeline **830** includes a media engine **837** and a video front end **834**. In one embodiment, the video front end **834** receives pipeline commands from the command streamer **803**. However, in one embodiment the media pipeline **830** includes a separate command streamer. The video front-end **834** processes media commands before sending the command to the media engine **837**. In one embodiment, the media engine includes thread spawning functionality to spawn threads for dispatch to the thread execution logic **850** via the thread dispatcher **831**.

In one embodiment, the graphics engine includes a display engine **840**. In one embodiment, the display engine **840** is external to the graphics processor and couples with the graphics processor via the ring interconnect **802**, or some other

15

interconnect bus or fabric. The display engine **840** includes a 2D engine **841** and a display controller **843**. The display engine **840** contains special purpose logic capable of operating independently of the 3D pipeline. The display controller **843** couples with a display device (not shown), which may be a system integrated display device, as in a laptop computer, or an external display device attached via an display device connector.

The graphics pipeline **820** and media pipeline **830** are configurable to perform operations based on multiple graphics and media programming interfaces and are not specific to any one application programming interface (API). In one embodiment, driver software for the graphics processor translates API calls that are specific to a particular graphics or media library into commands that can be processed by the graphics processor. In various embodiments, support is provided for the Open Graphics Library (OpenGL) and Open Computing Language (OpenCL) supported by the Khronos Group, the Direct3D library from the Microsoft Corporation, or, in one embodiment, both OpenGL and D3D. Support may also be provided for the Open Source Computer Vision Library (OpenCV). A future API with a compatible 3D pipeline would also be supported if a mapping can be made from the pipeline of the future API to the pipeline of the graphics processor.

FIG. **16A** is a block diagram illustrating a graphics processor command format according to an embodiment and FIG. **16B** is a block diagram illustrating a graphics processor command sequence according to an embodiment. The solid lined boxes in FIG. **16A** illustrate the components that are generally included in a graphics command while the dashed lines include components that are optional or that are only included in a sub-set of the graphics commands. The exemplary graphics processor command format **900** of FIG. **16A** includes data fields to identify a target client **902** of the command, a command operation code (opcode) **904**, and the relevant data **906** for the command. A sub-opcode **905** and a command size **908** are also included in some commands.

The client **902** specifies the client unit of the graphics device that processes the command data. In one embodiment, a graphics processor command parser examines the client field of each command to condition the further processing of the command and route the command data to the appropriate client unit. In one embodiment, the graphics processor client units include a memory interface unit, a render unit, a 2D unit, a 3D unit, and a media unit. Each client unit has a corresponding processing pipeline that processes the commands. Once the command is received by the client unit, the client unit reads the opcode **904** and, if present, sub-opcode **905** to determine the operation to perform. The client unit performs the command using information in the data **906** field of the command. For some commands an explicit command size **908** is expected to specify the size of the command. In one embodiment, the command parser automatically determines the size of at least some of the commands based on the command opcode. In one embodiment commands are aligned via multiples of a double word.

The flow chart in FIG. **16B** shows a sample command sequence **910**. In one embodiment, software or firmware of a data processing system that features an embodiment of the graphics processor uses a version of the command sequence shown to set up, execute, and terminate a set of graphics operations. A sample command sequence is shown and described for exemplary purposes, however embodiments are not limited to these commands or to this command sequence. Moreover, the commands may be issued as batch of com-

16

mands in a command sequence, such that the graphics processor will process the sequence of commands in an at least partially concurrent manner.

The sample command sequence **910** may begin with a pipeline flush command **912** to cause any active graphics pipeline to complete the currently pending commands for the pipeline. In one embodiment, the 3D pipeline **922** and the media pipeline **924** do not operate concurrently. The pipeline flush is performed to cause the active graphics pipeline to complete any pending commands. In response to a pipeline flush, the command parser for the graphics processor will pause command processing until the active drawing engines complete pending operations and the relevant read caches are invalidated. Optionally, any data in the render cache that is marked 'dirty' can be flushed to memory. A pipeline flush command **912** can be used for pipeline synchronization or before placing the graphics processor into a low power state.

A pipeline select command **913** is used when a command sequence requires the graphics processor to explicitly switch between pipelines. A pipeline select command **913** is required only once within an execution context before issuing pipeline commands unless the context is to issue commands for both pipelines. In one embodiment, a pipeline flush command is **912** is required immediately before a pipeline switch via the pipeline select command **913**.

A pipeline control command **914** configures a graphics pipeline for operation and is used to program the 3D pipeline **922** and the media pipeline **924**. The pipeline control command **914** configures the pipeline state for the active pipeline. In one embodiment, the pipeline control command **914** is used for pipeline synchronization and to clear data from one or more cache memories within the active pipeline before processing a batch of commands.

Return buffer state commands **916** are used to configure a set of return buffers for the respective pipelines to write data. Some pipeline operations require the allocation, selection, or configuration of one or more return buffers into which the operations write intermediate data during processing. The graphics processor also uses one or more return buffers to store output data and to perform cross thread communication. The return buffer state **916** includes selecting the size and number of return buffers to use for a set of pipeline operations.

The remaining commands in the command sequence differ based on the active pipeline for operations. Based on a pipeline determination **920**, the command sequence is tailored to the 3D pipeline **922** beginning with the 3D pipeline state **930**, or the media pipeline **924** beginning at the media pipeline state **940**.

The commands for the 3D pipeline state **930** include 3D state setting commands for vertex buffer state, vertex element state, constant color state, depth buffer state, and other state variables that are to be configured before 3D primitive commands are processed. The values of these commands are determined at least in part based the particular 3D API in use. 3D pipeline state **930** commands are also able to selectively disable or bypass certain pipeline elements if those elements will not be used.

The 3D primitive **932** command is used to submit 3D primitives to be processed by the 3D pipeline. Commands and associated parameters that are passed to the graphics processor via the 3D primitive **932** command are forwarded to the vertex fetch function in the graphics pipeline. The vertex fetch function uses the 3D primitive **932** command data to generate vertex data structures. The vertex data structures are stored in one or more return buffers. The 3D primitive **932** command is used to perform vertex operations on 3D primi-

17

tives via vertex shaders. To process vertex shaders, the 3D pipeline **922** dispatches shader execution threads to graphics processor execution units.

The 3D pipeline **922** is triggered via an execute **934** command or event. In one embodiment a register write triggers command execution. In one embodiment execution is triggered via a 'go' or 'kick' command in the command sequence. In one embodiment command execution is triggered using a pipeline synchronization command to flush the command sequence through the graphics pipeline. The 3D pipeline will perform geometry processing for the 3D primitives. Once operations are complete, the resulting geometric objects are rasterized and the pixel engine colors the resulting pixels. Additional commands to control pixel shading and pixel back end operations may also be included for those operations.

The sample command sequence **910** follows the media pipeline **924** path when performing media operations. In general, the specific use and manner of programming for the media pipeline **924** depends on the media or compute operations to be performed. Specific media decode operations may be offloaded to the media pipeline during media decode. The media pipeline can also be bypassed and media decode can be performed in whole or in part using resources provided by one or more general purpose processing cores. In one embodiment, the media pipeline also includes elements for general-purpose graphics processor unit (GPGPU) operations, where the graphics processor is used to perform SIMD vector operations using computational shader programs that are not explicitly related to the rendering of graphics primitives.

The media pipeline **924** is configured in a similar manner as the 3D pipeline **922**. A set of media pipeline state commands **940** are dispatched or placed into in a command queue before the media object commands **942**. The media pipeline state commands **940** include data to configure the media pipeline elements that will be used to process the media objects. This includes data to configure the video decode and video encode logic within the media pipeline, such as encode or decode format. The media pipeline state commands **940** also support the use one or more pointers to "indirect" state elements that contain a batch of state settings.

Media object commands **942** supply pointers to media objects for processing by the media pipeline. The media objects include memory buffers containing video data to be processed. In one embodiment, all media pipeline state must be valid before issuing a media object command **942**. Once the pipeline state is configured and media object commands **942** are queued, the media pipeline **924** is triggered via an execute **934** command or an equivalent execute event (e.g., register write). Output from the media pipeline **924** may then be post processed by operations provided by the 3D pipeline **922** or the media pipeline **924**. In one embodiment, GPGPU operations are configured and executed in a similar manner as media operations.

FIG. 17 illustrates exemplary graphics software architecture for a data processing system according to an embodiment. The software architecture includes a 3D graphics application **1010**, an operating system **1020**, and at least one processor **1030**. The processor **1030** includes a graphics processor **1032** and one or more general-purpose processor core (s) **1034**. The graphics application **1010** and operating system **1020** each execute in the system memory **1050** of the data processing system.

In one embodiment, the 3D graphics application **1010** contains one or more shader programs including shader instructions **1012**. The shader language instructions may be in a high-level shader language, such as the High Level Shader Language (HLSL) or the OpenGL Shader Language (GLSL).

18

The application also includes executable instructions **1014** in a machine language suitable for execution by the general-purpose processor core **1034**. The application also includes graphics objects **1016** defined by vertex data.

The operating system **1020** may be a Microsoft® Windows® operating system from the Microsoft Corporation, a proprietary UNIX-like operating system, or an open source UNIX-like operating system using a variant of the Linux kernel. When the Direct3D API is in use, the operating system **1020** uses a front-end shader compiler **1024** to compile any shader instructions **1012** in HLSL into a lower-level shader language. The compilation may be a just-in-time compilation or the application can perform share pre-compilation. In one embodiment, high-level shaders are compiled into low-level shaders during the compilation of the 3D graphics application **1010**.

The user mode graphics driver **1026** may contain a back-end shader compiler **1027** to convert the shader instructions **1012** into a hardware specific representation. When the OpenGL API is in use, shader instructions **1012** in the GLSL high-level language are passed to a user mode graphics driver **1026** for compilation. The user mode graphics driver uses operating system kernel mode functions **1028** to communicate with a kernel mode graphics driver **1029**. The kernel mode graphics driver **1029** communicates with the graphics processor **1032** to dispatch commands and instructions.

To the extent various operations or functions are described herein, they can be described or defined as hardware circuitry, software code, instructions, configuration, and/or data. The content can be embodied in hardware logic, or as directly executable software ("object" or "executable" form), source code, high level shader code designed for execution on a graphics engine, or low level assembly language code in an instruction set for a specific processor or graphics core. The software content of the embodiments described herein can be provided via an article of manufacture with the content stored thereon, or via a method of operating a communication interface to send data via the communication interface.

A non-transitory machine readable storage medium can cause a machine to perform the functions or operations described, and includes any mechanism that stores information in a form accessible by a machine (e.g., computing device, electronic system, etc.), such as recordable/non-recordable media (e.g., read only memory (ROM), random access memory (RAM), magnetic disk storage media, optical storage media, flash memory devices, etc.). A communication interface includes any mechanism that interfaces to any of a hardwired, wireless, optical, etc., medium to communicate to another device, such as a memory bus interface, a processor bus interface, an Internet connection, a disk controller, etc. The communication interface is configured by providing configuration parameters or sending signals to prepare the communication interface to provide a data signal describing the software content. The communication interface can be accessed via one or more commands or signals sent to the communication interface.

Various components described can be a means for performing the operations or functions described. Each component described herein includes software, hardware, or a combination of these. The components can be implemented as software modules, hardware modules, special-purpose hardware (e.g., application specific hardware, application specific integrated circuits (ASICs), digital signal processors (DSPs), etc.), embedded controllers, hardwired circuitry, etc. Besides what is described herein, various modifications can be made to the disclosed embodiments and implementations of the invention without departing from their scope. Therefore, the

illustrations and examples herein should be construed in an illustrative, and not a restrictive sense. The scope of the invention should be measured solely by reference to the claims that follow.

The following clauses and/or examples pertain to further embodiments:

One example embodiment may be a method comprising storing depth values in a depth buffer for multi-sampled anti-aliasing, said buffer having a number of samples per pixel, and coding the depth values so that the size of the depth buffer is the same regardless of the number of samples per pixel. The method may also include dividing a frame into groups of pixels of a predetermined size. The method may also include determining whether all the samples of a group are lit. The method may also include wherein if all the samples of the group are not lit, coding the minimum/maximum depth values to represent the group. The method may also include wherein if all the pixels of the group are lit, coding the group using its plane equation. The method may also include coding the depth values of the group of pixels differently depending on whether all the samples of the group are lit. The method may also include coding a plane bit, a pixel mask and a minimum and maximum depths of a group of pixels if not all the samples of the group are lit, encoding the plane bit, pixel mask and plane equation for the group of pixels if all the samples of the group are lit.

Another example embodiment may be one or more non-transitory computer readable media storing instructions executed by a processor to perform a sequence comprising storing depth values in a depth buffer for multi-sampled anti-aliasing, said buffer having a number of samples per pixel, and coding the depth values so that the size of the depth buffer is the same regardless of the number of samples per pixel. The media may include said sequence including dividing a frame into groups of pixels of a predetermined size. The media may include said sequence including determining whether all the samples of a group are lit. The media may include said sequence including wherein if all the samples of the group are not lit, coding the minimum/maximum depth values to represent the group. The media may include wherein if all the pixels of the group are lit, coding the group using its plane equation. The media may include said sequence including coding the depth values of the group of pixels differently depending on whether all the samples of the group are lit. The media may include said sequence including coding a plane bit, a pixel mask and a minimum and maximum depths of a group of pixels if not all the samples of the group are lit, encoding the plane bit, pixel mask and plane equation for the group of pixels if all the samples of the group are lit.

In another example embodiment may be an apparatus comprising a hardware processor to store depth values in a depth buffer for multi-sampled anti-aliasing, said buffer having a number of samples per pixel and code the depth values so that the size of the depth buffer is the same regardless of the number of samples per pixel, and a storage coupled to said processor. The apparatus may include said processor to divide a frame into groups of pixels of a predetermined size. The apparatus may include said processor to determine whether all the samples of a group are lit. The apparatus may include wherein if all the samples of the group are not lit, coding the minimum/maximum depth values to represent the group. The apparatus may include wherein if all the pixels of the group are lit, said processor to code the group using its plane equation. The apparatus may include said processor to code the depth values of the group of pixels differently depending on whether all the samples of the group are lit. The apparatus may include said processor to code a plane bit, a pixel mask

and a minimum and maximum depths of a group of pixels if not all the samples of the group are lit, encoding the plane bit, pixel mask and plane equation for the group of pixels if all the samples of the group are lit.

The graphics processing techniques described herein may be implemented in various hardware architectures. For example, graphics functionality may be integrated within a chipset. Alternatively, a discrete graphics processor may be used. As still another embodiment, the graphics functions may be implemented by a general purpose processor, including a multicore processor.

References throughout this specification to “one embodiment” or “an embodiment” mean that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one implementation encompassed within the present disclosure. Thus, appearances of the phrase “one embodiment” or “in an embodiment” are not necessarily referring to the same embodiment. Furthermore, the particular features, structures, or characteristics may be instituted in other suitable forms other than the particular embodiment illustrated and all such forms may be encompassed within the claims of the present application.

While a limited number of embodiments have been described, those skilled in the art will appreciate numerous modifications and variations therefrom. It is intended that the appended claims cover all such modifications and variations as fall within the true spirit and scope of this disclosure.

What is claimed is:

1. A method comprising:

storing depth values in a depth buffer for multi-sampled anti-aliasing, said buffer having a number of samples per pixel;

coding the depth values using a hardware processor so that the amount of storage used to store samples is not dependent on how many samples are used;

dividing using a hardware processor a frame into groups of pixels of a predetermined size;

determining using a hardware processor whether all the samples of a group are lit;

coding the depth values of the group of pixels differently depending on whether all the samples of the group are lit; and

rendering the lit samples for computer display.

2. The method of claim 1 including wherein if all the samples of the group are not lit, coding minimum/maximum depth values to represent the group.

3. The method of claim 1 wherein if all the pixels of the group are lit, coding the group using its plane equation.

4. The method of claim 1 including coding a plane bit, a pixel mask and a minimum and maximum depths of a group of pixels if not all the samples of the group are lit, encoding the plane bit, pixel mask and plane equation for the group of pixels if all the samples of the group are lit.

5. One or more non-transitory computer readable media storing instructions executed by a processor to perform a sequence comprising:

storing depth values in a depth buffer for multi-sampled anti-aliasing, said buffer having a number of samples per pixel; and

coding the depth values so that the amount of storage used to store samples is not dependent on how many samples are used;

dividing a frame into groups of pixels of a predetermined size; determining whether all the samples of a group are lit; and

21

coding the depth values of the group of pixels differently depending on whether all the samples of the group are lit; and

rendering the lit samples for computer display.

6. The media of claim 5, said sequence including wherein 5
if all the samples of the group are not lit, coding minimum/
maximum depth values to represent the group.

7. The media of claim 5 wherein if all the pixels of the group are lit, coding the group using its plane equation.

8. The media of claim 5, said sequence including coding a 10
plane bit, a pixel mask and a minimum and maximum depths
of a group of pixels if not all the samples of the group are lit,
encoding the plane bit, pixel mask and plane equation for the
group of pixels if all the samples of the group are lit.

9. An apparatus comprising:

15 a hardware processor to store depth values in a depth buffer
for multi-sampled anti-aliasing, said buffer having a
number of samples per pixel and code the depth values
so that the amount of storage used to store samples is not
dependent on how many samples are used; said processor

22

to divide a frame into groups of pixels of a predetermined size, determine whether all the samples of a group are lit, and code the depth values of the group of pixels differently depending on whether all the samples of the group are lit, and render the lit samples for computer display; and

a storage coupled to said processor.

10. The apparatus of claim 9 wherein if all the samples of the group are not lit, coding minimum/maximum depth values to represent the group.

11. The apparatus of claim 9 wherein if all the pixels of the group are lit, said processor to code the group using its plane equation.

12. The apparatus of claim 9, said processor to code a plane bit, a pixel mask and a minimum and maximum depths of a group of pixels if not all the samples of the group are lit, encoding the plane bit, pixel mask and plane equation for the group of pixels if all the samples of the group are lit.

* * * * *